Programming the M68000

68000プログラミング入門

Tim King, Brian Knight 共著　鈴木 隆監訳

アスキー出版局

68000プログラミング入門

Tim King, Brian Knight 共著　鈴木 隆監訳

アスキー出版局

Programming the M68000

Tim King and Brian Knight

(ISBN 0-201-11730-4)

Illustrated by Fumitaka Kuroda
Cooperation: Akio Kambayashi
Studio V

目 次

著者まえがき …… 7
監訳者まえがき …… 9

第1章 イントロダクション 11

1.1 マイクロプロセッサの発展……13
1.2 68000の概要……21
1.3 位置独立コード……25
1.4 チップが提供するデバッグ支援機能……26
1.5 高級言語のサポート……28
1.6 オペレーティング・システムのサポート……30
1.7 典型的なアプリケーション……32
1.8 68000シリーズのプロセッサ……32

第2章 アセンブラの構文とアドレッシングモード 37

2.1 アセンブラの構文……38
2.2 アセンブラ・ディレクティブ……41
2.3 アセンブラの構文のまとめ……45
2.4 式……46
2.5 アドレッシングモード……48
2.6 実効アドレスの分類……61

第3章 データの移動と比較 65

3.1 単純なデータ移動……66
3.2 条件付き分岐(1)……69
3.3 比較……72

3.4　条件付き分岐(2)……………75
3.5　簡単なメモリ・チェック例……………77
3.6　ゼロとの比較と移動……………79
3.7　小さい数の移動……………80
3.8　ビットテスト……………81
3.9　条件テスト……………82
3.10 ループ制御……………83
3.11 簡単な入出力……………86
3.12 周辺装置へのデータ移動……………89

第4章　スタックとサブルーチン　93

4.1　サブルーチン……………99
4.2　アブソリュート・ジャンプ……………106
4.3　実効アドレス……………109
4.4　スタック領域の割付け……………112
4.5　メモリ診断プログラム……………116

第5章　算術演算　123

5.1　加算……………124
5.2　減算……………126
5.3　値の負数を取る……………127
5.4　乗算……………128
5.5　レジスタ値の交換……………129
5.6　倍長の乗算……………130
5.7　除算……………135

5.8 倍長の除算……………136
5.9 10進演算……………139

第6章 論理演算 147

6.1 シフトとローテイト……………151
6.2 16進表現への変換……………154
6.3 単一ビットの演算……………155
6.4 フリーエリア割付けパッケージ……………157

第7章 例外処理 163

7.1 例外処理ベクタ……………165
7.2 ユーザーモードとスーパーバイザモード……………167
7.3 例外処理の動作……………168
7.4 例外処理ルーチン……………169
7.5 割込み……………172
7.6 外部リセット……………173
7.7 不正命令と未実装命令……………173
7.8 トラップの原因となる命令……………174
7.9 特権違反……………176
7.10 トレース……………178
7.11 バスエラーとアドレスエラー……………179
7.12 例外処理の順位付け……………181
7.13 メモリサイズ判定ルーチン……………182

第8章　モニタ・プログラム　187

8.1　定数の定義……………191
8.2　入出力……………194
8.3　分岐テーブル……………201
8.4　初期設定とコマンド……………203
8.5　単純なコマンド・ルーチン……………205
8.6　レジスタの表示と更新……………207
8.7　ユーザー・プログラムの実行……………212
8.8　メモリの確認・更新ルーチン……………216
8.9　ブレーク・ポイント……………220
8.10　例外処理手続き……………222
8.11　メッセージとテーブル……………229

付　録　233

68000命令セット……………233
条件テスト……………241

索　引　242

著者まえがき

本書は最初から最後まで，通読できるような構成になっています．全編を読みとおすことで，68000アセンブリ言語の初歩的なプログラミングをすべて御理解いただけます．より経験を積んだ読者の方々のために，付録に命令セットの表を掲載しています．この表では，個々の命令の簡単な説明を示すとともに，本文で詳細に説明している箇所のページも示しています．

68000の後続機種である68010と68020に関する情報は，モトローラによる暫定情報から引用したものです．モトローラに対し，これらの情報の提供に感謝するとともに，68000自体のドキュメントからの引用を認めてくださったことに感謝の意を表します．

モトローラは，本書での記載上の誤りに関して，何らの責任を負うものではなく，また記載されている製品に関し，信頼性，機能，または設計上の改善のために，変更を加える権利を有するものとします．モトローラは，アプリケーション，または本書に記載されている製品の使用に起因する障害に関し，何らの責任も負わないものとします．いかなる形式でも特許権の基に使用許諾権は譲渡されません．新製品の仕様は，予告なく変更される場合があります．

本書の執筆にあたって，Universities of Cambridge and Bath の同僚たちの協力を得，また特に，Dr. Arthur Norman が倍長の除算ルーチンの使用を許可してくださったことに感謝の意を表したいと思います．また，表の作図をしてくださった Agi Lehar-Graham，および索引の作成を手伝ってくださった Jessica King の両氏にも感謝いたします．

1983年3月　Tim King
Brian Knight

監訳者まえがき

本書は68000のアセンブリ・プログラミングを学ぶ人のために書かれています．従来，こういった分野のものは，どちらかというと各命令の紹介に重点が置かれ，命令の羅列に終始してしまいがちでした．

本書では命令個々の説明は必要最小限に止めており，また説明に際しても具体性を失わないよう，注意深くプログラム例が選択されています．加えて，読者の方々が自分でプログラミングする際に，それを部品として再利用することもできるように，細かい配慮がなされています．

プログラム例としては，フリーエリア割付けパッケージや割込み処理など，興味深いテーマが多くみられ，本文の解説やプログラム自身を分析することによって，プログラミング・スタイル，およびプログラミング・テクニックなど，多くのものを学び取れることでしょう．

本書のプログラム例の中でも特徴的なものとして，第8章で述べられている小型のモニタ・プログラムがあります．このモニタは実用性を考えると，その機能はいささか不充分なものですが，68000を理解する上で必要とされることの大部分は網羅されています．一例を挙げると，割込みや例外処理の取扱い，条件分岐命令や各種アドレッシングモードの適切な使用法など，断片的なプログラム例だけでは充分に説明しきれない部分が多く含まれています．

このモニタ・プログラムは，約2Kバイトと扱いやすい大きさに抑えられていますので，時間をかけてじっくり読んでいけば，容易にその全貌をつかめることと思います．

8ビットはある程度理解できたけれども，16ビットとなるとなにやら難しそうだ，と思って敬遠していた読者は，本書で示されているプログラム例をみて意外な印象を受けるのではないでしょうか．

70年代半ばに登場したマイクロプロセッサの性能は非常に限られたものでした．アプリケーションの種類によっては，好むと好まざるとに関わらず，いわゆるテクニックと呼ばれる（人前では決して自慢してはいけない），あまりエレガントとは言えないプログラミングを強いられることがしばしばありました．

ところが，68000のように洗練された命令体系を備え，充分な性能を備えたマイクロプロセッサでは，シンプルで強力な命令を素直に使用することによって，自然なプログラミングを行うことができるのです．ある意味では，8ビット・プロセッサ（例：Z80）よりも扱いが簡単かもしれません．というのも，各プロセッサ固有の瑣末な制約がより少なくなっている為に，プログラマはプログラムの本質的な部分により専念できると考えられるからです．命令のバイト長やクロック数を常に気に掛けながらのプログラミングはもう昔話になるべきだとは思いませんか!?

16ビット・プロセッサの本命として，色々な面で注目を集めている68000ですが，現時点では高級なワーク・ステーションや医療機器，画像処理など，比較的限られた範囲にその応用は限定されています．68000が搭載されているパーソナル・コンピュータは，残念ながら，Apple 社の Macintosh, Lisa や HP 社の9000シリーズなど数える程でしか無く，個人で購入するにはいささか高価すぎます．しかし，過去10年間のパーソナル・コンピュータの流れを考えて見れば，ミニコンピュータ並のパワーがパソコン・クラスの価格で使えるようになる時代は，もう，すぐそこまで来ていると断言できるでしょう．

訳出に当たっては，この種の翻訳書としては珍しく，プログラム中のコメント部分を原文のまま引用しています．コメント部分を日本語に訳すかどうかについては，議論の分かれるところですが，職業的プログラマを除くと，実際のコメント例を見る機会は意外に少ないのではないでしょうか．この意味において，初心者にとって良い参考資料になると考え，敢えて原文のまま引用しています．また，必要に応じて訳注を付け加えました．読者の方々の理解の助けになれば幸いに思います．なお，訳注は本文に注マークを付け，各章末にまとめて示してあります．

本書の監訳にあたって翻訳者の三浦明美さん，ならびにリストの打ち込みを手伝ってくれた坂中二郎君にいろいろお世話になりました．ここに感謝の意を表します．

1984年10月　鈴木　隆

CHAPTER 1

イントロダクション

1.1	マイクロプロセッサの発展	13
1.2	68000の概要	21
1.3	位置独立コード	25
1.4	チップが提供するデバッグ支援機能	26
1.5	高級言語のサポート	28
1.6	オペレーティング・システムのサポート	30
1.7	典型的なアプリケーション	32
1.8	68000シリーズのプロセッサ	32

はじめに

本書は題名が示すとおり，プログラマの立場から見た68000マイクロプロセッサを中心に説明しています．したがってハードウェア関連の事項についてはほとんど省略しています．本書の読者対象は，68000システムに触れることができ，効果的なプログラミングに興味を持っている人々です．

68000のアーキテクチャ，およびその命令セットを論理的順序で説明していますので，本書により68000のプログラミングを総合的に理解できます．モトローラ社の68000のマニュアル[注1]を参照しなくても読み進むことができますが，本書の後半部分において，例えば各命令のビット・パターンなどの詳しい点については，マニュアルを参照する必要があります．

個々の命令の説明では，命令の特徴および注意すべき点とその使用方法について指摘します．これらの特徴は，形式的に定義を読んでいるときには容易に把握できますが，実際に使用する段になると，時間を浪費し混乱を起こす原因になりがちです．個々の命令を紹介するごとに，その命令の使用法を理解しやすくするため，実際のプログラミング例をいくつか示します．これらのプログラミング例は，より大きなプログラムの中にも組み入れられる，便利なコードとなっています．また，これらのプログラミング例は，本書では，簡単な入出力とデバッグ機能を提供する，小型のモニタ・プログラムを構成する目的で使用されています．

第1章では，マイクロプロセッサの発展の歴史を簡単に説明し，68000と現在使用されている他のマイクロプロセッサを比較します．次に68000の機能について概要を説明し，代表的なアプリケーションをいくつか示します．

第2章では，第3章以降で使用するアセンブラの構文について紹介するとともに，命令のオペランド・アドレッシングモードを説明します．

第3章以降は，個々の命令を関連性のあるグループに分けて説明していきます．

第3章では，データを移動・比較するさまざまな方法について．

第4章では，スタックとサブルーチンの概念について．

第5章では，算術演算のための命令と，直接取り扱うことのできない大きな

数の乗算と除算ルーチンについて．

第6章では，個々のビットに対して作用する論理演算（フリーエリア割付けパッケージのコーディングに使用される）について．

第7章では，割込みとトラップについて説明します．特に割込みルーチンの書き方とエラー検出，プログラムのデバッグ用のシステム・コールとしてのトラップの使い方を示します．

最後の第8章では，小型のモニタ・プログラムの完全な例を示します．このモニタは端末装置と入出力を行い，プログラムをデバッグするための便利な環境を提供します．

1.1 マイクロプロセッサの発展

40数年にわたるコンピュータの歴史の中で，一貫して変わらない傾向として，コンピュータがますます小型化されつつある，ということが言えます．最も初期のコンピュータは真空管を使用していたため，広い空間を必要とし，大量の電力を消費していました．その後，トランジスタの発明により，コンピュータの大きさと電力消費量は何分の一かに減らすことが可能になりました．

1960年代に入ると，1枚の小さなシリコンチップに幾つかのトランジスタと，関連する素子を組み込んだ集積回路の生産が可能になり，コンピュータは取り扱い易いキャビネットサイズのものになりました．

1970年代初期には，集積回路の技術が進歩し，簡単なコンピュータの中央処理装置のすべてを1枚のチップに乗せることができるようになりました．これが最初のマイクロプロセッサです．それ以来，マイクロプロセッサはめざましく進歩してきました．現在，1枚のチップに乗せるために性能を犠牲とせずに複数の個別部品から構成されるコンピュータと競争しうるマイクロプロセッサが使用可能になっています．

初期の段階で一般に使用されていたマイクロプロセッサは，一度に4ビットのデータしか扱えないものでした（例：インテル4004）．このようなマイクロプロセッサは，単純な制御アプリケーション（例：自動販売機，警報システムなど）や洗練度の低いビデオゲームには適していましたが，使い易い単位でのデータ

を取り扱う場合には，あまりにも遅く不便でした．また扱えるメモリ量も非常に限られたものでした．

マイクロプロセッサが広く使われ出したのは，8ビットマシンの登場以後のことです．8ビットマシンの代表的機種としては，インテル8080，8085，ザイログZ80(命令セットのサブセットとして8080の命令を使用)，MOSテクノロジー6502，そしてモトローラ6800，6809があります．〝**Nビットプロセッサ**〞と呼ばれるマシンの場合，2Nビットのサイズのデータを処理できる機能を持っているのが普通です．上記の機種の大半は，16ビットのデータに対する算術および論理演算を実行できます．ただし，6502は16ビットの内部レジスタを持ちません．

これらの8ビットマシンのいくつかは非常に安価になり，他の装置に組み込まれたり，中程度の機能と低価格を特徴とするホームコンピュータにも使用されるようになりました．本書の執筆時点では，家庭用およびスモールビジネス用のパーソナル・コンピュータの多くはZ80または6502を基盤としています．

8ビット・マイクロプロセッサが発展するにつれて，マイクロプロセッサの持つ〝不便さ〞――すなわち，いくつかの異なる電圧，多相クロック入力，多重化されたアドレス/データ線など――が克服されつつあります．現在，新しい設計の機種では，単一の5V電源，単相クロック入力(または内部クロック制御用のクリスタルのみを外付けする)，および各接続ピンによる単一の機能の実行が普通になっています．いくつかの機種(例：Z80)では，ダイナミック型半導体メモリ(**DRAM**)のリフレッシュ機能も提供されています．

開発のもう一つの成果としてTMS9940などの限定型シングルチップ・コンピュータがあります．これらのコンピュータには，プロセッサの他に，デバッグ済みのプログラムを入れるメモリ(**ROM**)と，書き替え可能なメモリ(**RAM**)があり，単一パッケージの特定用途向けコンピュータとなっています．これにより，他の回路との接続が容易です．この種の装置は，機器の中の特定部分として設計したり，あるいはその部分に組み込む目的に最も適しています．

ソフトウェアの点から考えた場合，1970年代の後半に登場した16ビットと32ビット・マイクロプロセッサが次に重要な開発結果と言えます．これらのマイクロプロセッサの登場により，ミニ・コンピュータとマイクロコンピュータの境界線は不明瞭になりました．というのも一般的なミニ・コンピュータの大半は16ビットマシンだからです．この種のチップの最初のものとしては，テキ

サス TMS9900シリーズ，インテル8086，テキサス TMS99000，ザイログ Z8000．そして本書で扱うモトローラ68000があります．現時点ではそれほど一般的でない新しいチップとして，ナショナル・セミコンダクタ NS32016，インテル iAPX286，および iAPX432があります．

68000は，大型のメインフレームと類似したアーキテクチャと命令セットを装備した最初のマイクロプロセッサであることから，従来のマイクロプロセッサとは一線を画しています．68000の特徴として非常に大きな直接アクセス可能なアドレス空間，8，16および32ビットのデータを処理する機能，各32ビット長の16個のレジスタ，高級言語のコンパイルを容易にするいくつかの命令，特権化されていないプログラムが，メモリ内の一定の領域をアクセスしたり，直接 I/O 動作の起動を禁止するためのスーパーバイザモード．そしてマルチプロセッサ間のインターロック機構の提供があります．

上記の各プロセッサを比較するため，以下に簡単な仕様を示します．ここで注意すべき点は異なるプロセッサの速度を比較する場合です．というのは，ある種のマシンの後期のモデルは，初期のモデルよりも速いクロック速度で作動可能だからです．したがって，プロセッサの速度は，設計上の能力を表すというよりもむしろ，そのマシンが市場に登場してからどれほど経過しているかを反映していると言えます．

▶ **MOS テクノロジー6502**

直接アドレス可能な領域：64K バイト
最短実行時間：0.5マイクロ秒（4MHz クロック）
汎用レジスタ：（8ビット）× 1
その他のレジスタ：8ビット・インデックス・レジスタ× 2
8ビット・スタック・ポインタ
割込みレベル：2

メモリの0ページ内のバイトは，16ビットのインデックス付けのために，対にすることができます．命令セットは，最良なアドレッシングモードを選択できるようになっていますが，8ビットより長いデータを直接取り扱うための命令はありません．

▶ザイログ Z80

直接アドレス可能な領域：64Kバイト
最短実行時間：1マイクロ秒(4MHz クロック)
汎用レジスタ：(8ビット)×7＋重複セット
その他のレジスタ：16ビット・インデックス・レジスタ×2
　　　　　　　　　16ビット・スタック・ポインタ
割込みレベル：2

8ビット・レジスタは対にすることにより，3個の16ビット・レジスタとして使用することができます．命令セットは，16ビットの算術演算，およびメモリ内のブロック転送とブロック・サーチをサポートしています．インテル8080の命令は Z80の命令のサブセットとなっています．

▶モトローラ6809

直接アドレス可能な領域：64Kバイト
最短実行時間：1マイクロ秒(2MHz クロック)
汎用レジスタ：2
その他のレジスタ：16ビット・インデックス・レジスタ×2
　　　　　　　　　16ビット・スタック・ポインタ×2
割込みレベル：3

2個の8ビット・アキュムレータは，16ビット・レジスタとして連結することができます．命令セットは，限定された16ビットの算術演算，および8×8ビットの乗算を行うことができます．

▶インテル8086

直接アドレス可能な領域：1Mバイト
最短実行時間：0.4マイクロ秒(5MHz クロック)
汎用レジスタ：(16ビット)×4
その他のレジスタ：セグメント・ベース・レジスタ×4

インデックス・レジスタ×2
ベース・レジスタ，スタック・ポインタ
割込みレベル：2

アドレス空間は4つのセグメント(コード，データ，スタックおよびエクストラ)に分かれており，これらはオーバーラップする場合もあります．アドレッシングはすべて，セグメント・ベース・レジスタ相対です(セグメント・ベース・アドレスは16の倍数)．8086には8のオペランド・アドレッシングモードがあり，符号付き/符号なし16ビット乗算/除算ができ，ループ命令があります．また，不可分なリード・モディファイ・ライト・メモリ・アクセスが可能です．インテル8088プロセッサは，8086と同じソフトウェアを実行できますが，8ビット(16ビットではない)の外部バスを持っているため，8ビット・サポートチップとともに使用することができます．

▶ **TMS9900シリーズ**
直接アドレス可能な領域：64Kバイト(TMS9900)
最短実行時間：0.5マイクロ秒(4MHz クロック)
汎用レジスタ：16(内部的にではなくRAM上に置かれる)
その他のレジスタ：ワークスペース・ポインタ(すなわち，レジスタ)
割込みレベル：16(TMS9900，9995)
4(その他)

レジスタは，ワークスペース・ポインタ・レジスタによって示されるRAMの中の領域に置かれています．TMS9995では，レジスタは内部的にキャッシュされています．16ビットの乗算/除算命令があります．
共通の命令セットを備えたプロセッサのファミリは，次のとおりです．

○9900………基本モデル
○9940………組込みRAMおよびROM付きシングルチップ・コンピュータ
○9980/81…8ビット・バスのみ．アドレス可能領域は16Kバイトのみ
○9985………組込みRAM付き(ROMなし)シングルチップ・コンピュータ

○9995………レジスタは内部的にキャッシュされている.

▶ TMS99000

直接アドレス可能な領域：64K バイト
最短実行時間：0.5マイクロ秒(6MHz クロック)
汎用レジスタ：16(RAM に収容)
その他のレジスタ：ワークスペース・ポインタ(レジスタに対するポインタ)
割込みレベル：16

99000はサポートチップにより，最高16M バイトのセグメント化されたメモリをアドレスすることができます．32ビットのデータの加算，減算およびシフトが可能です．スーパーバイザモードがあり，また複数のプロセッサ間の同期をとるためのテスト・アンド・セット命令があります．命令のデコードの方法として，組み込まれていない命令コードにおいては，ユーザーのマイクロコード(チップに搭載)，外部 RAM 内のユーザーコード，または接続されたプロセッサによって処理することができます．

▶ザイログ Z8000

直接アドレス可能な領域：8M バイト(Z8001)
最短実行時間：0.5マイクロ秒(8MHz クロック)
汎用レジスタ：(16ビット)×16
その他のレジスタ：メモリ・リフレッシュ・カウンタ
　　　　　　　　　ステータス・エリア・ポインタ
割込みレベル：2

6つのアドレス空間[注2]があり，それぞれ8Mバイトの大きさです．チップには，8Mバイトのアドレス可能領域を持つ〝セグメント化された〞チップ(Z8001)[注3]と，64Kバイトのアドレス可能領域を持つ〝セグメント化されていない〞チップ(Z8002)の2つのバージョンがあります．先頭の8個のレジスタは，16個の8ビット・レジスタとして使用することができます．レジスタ群は，16個の16ビット・レジスタ，8個の32ビット・レジスタ，または4個の64ビット・レジス

タとして使用することができます．乗算は16ビットまたは32ビットのオペランドに対して使用可能です．除算は32ビットまたは64ビットのオペランドに対して使用可能です．シフトは，8，16または32ビット・レジスタに対して実行することができます．その他スーパーバイザモード，テスト・アンド・セット命令，および複数のプロセッサ間でインタフェースをとる命令があります．Z8000には，ブロック・コピー，および文字変換の命令があります．アドレス・モードは8個ありますが，プロセッサ・トラップは4タイプのみです．

▶モトローラ68000

直接アドレス可能な領域：16Mバイト
最短実行時間：0.5マイクロ秒(8MHz クロック)
汎用レジスタ：(32ビット)×16
その他のレジスタ：ユーザー・スタック・ポインタ
割込みレベル：7

内部アーキテクチャは32ビット幅で，大半の処理は8，16または32ビット値に対して実行することができます．ただし32ビットの乗算および除算機能がない点だけ，完全な32ビット機能から欠落しています．またアドレス空間は線形です．レジスタは8個のデータ・レジスタと8個のアドレス・レジスタに分類され，処理によっては，このうち一方のタイプのレジスタしか使用できない場合もあります．アドレス・レジスタのうち1個は重複しています．どちらを使用できるかは，プロセッサがスーパーバイザまたはユーザーモードのどちらにあるかによって決まります．14個のオペランド・アドレッシングモード，さまざまなタイプのプロセッサ・トラップおよびスーパーバイザモードでのみ使用可能な命令があります．リード・モディファイ・ライト・メモリ・アクセスのために，〝テスト・アンド・セット〞命令が準備されています．

▶ NS32016

直接アドレス可能な領域：16M バイト
最短実行時間：0.3マイクロ秒(10MHz クロック)
汎用レジスタ：(32ビット)×8
その他のレジスタ：スタティック・ベース，フレーム・ポインタ，
　　　　　　　　　スタック・ポインタ×2，モジュール，
　　　　　　　　　インタラプト・ベース
割込みレベル：2

32016は内部的には完全な32ビット・プロセッサですが，外部バスのみは16ビット幅です．高級言語(Algol, Pascal など)を効率よくサポートするために命令の高機能化が計られています．FPU や MMU の様なコ・プロセッサのための命令を最初から用意されており，CPU のレジスタと同様にアクセスできます．1，2，4バイトの整数，および4，8バイトの実数についてほぼ完全に同等の命令が用意されています．またアドレッシングについては直交性が非常に高く，マイクロプロセッサの中では最も VAX に近いアーキテクチャになっています．MMU はデマンドページ方式の仮想記憶方式をサポートしています．また，高級言語のモジュールの概念を直接サポートするために，ソフトウェア・モジュール(コード，静的，動的，および外部変数の完全な分離)をサポートしています．このため，アドレス・レジスタは専用レジスタとして実現され，8本の汎用レジスタはデータ・レジスタとしてだけでなく，ポインタとしても使用できます．割込みテーブルがレジスタによって指定されるため，任意の番地に置け，仮想マシンの実現も容易です．

▶インテル iAPX286

直接アドレス可能な領域：16M バイト
最短実行時間：0.25マイクロ秒(8MHz クロック)
汎用レジスタ：(16ビット)×8
割込みレベル：2

iAPX286は8086/8088と上位互換性があり，これらの機種用のプログラムはほ

とんど変更することなく(あるいはまったく変更なしで)実行することができます．相違点は速度と保護されたマルチユーザー・システムに対するサポート機能にあります．メモリ管理および保護機能はプロセッサチップに含まれており，外部でのメモリ管理は不要です．命令のほとんどは，例外処理後に再実行可能であり，最高1ギガバイト(1000メガバイト)の仮想記憶を提供することができます．割込み処理後に，オペレーティング・システムによる介入なしに即時にタスク切替えを実行する，ハードウェアによるサポート機能があります．

▶インテルiAPX432システム(暫定情報より)

iAPX432システムのプロセッサは，43201命令デコーダと，43202実行ユニットの，2つのチップから構成されます．I/Oは43203インタフェース・プロセッサにより処理されます．

データは最高32ビット単位で処理され，最高80ビット長の浮動小数点数がサポートされています．アドレッシングはケーパビリティに基づいて行われ，個々のデータ構造に対して保護機能を適用できます．アドレス可能な実記憶は最高16メガバイトですが，ソフトウェアは，最高1テラバイト(1000ギガバイト)の仮想アドレス空間を使用できます．

複数のプロセッサ，マルチタスキング，および動的記憶域割当てのための組込みサポート機能があります．命令長は，2，3のビットから数百ビットまで変化することができ，また命令の開始位置を特定のメモリ境界に揃える必要がないという特殊な利点があります．2個のプロセッサを並列に接続することにより，一方のプロセッサで他方の動作のチェックを行うことで，システムの信頼性が高められます．

1.2 68000の概要

本節から第2章以降に記載されている詳しい説明の予備知識として，68000の概要を述べていきます．個々の命令については，ごく簡単な説明だけで済ます場合もありますが，これは，他のコンピュータのアセンブリ言語をすでに知っている読者を意識したためです．初心者の方で理解できない点があると思いま

すが，本書の後半に詳しい説明がありますので心配する必要はありません．

68000で使用可能なメモリには，内部の**レジスタ**（チップに搭載）と外部の**主記憶**の2種類があります．レジスタは全部で17個ありますが，そのうちいつでも使用可能なのは16個だけです．この中の8個が**データ・レジスタ**であり，D0～D7という名前がついています．その他が**アドレス・レジスタ**で，A0～A7と呼ばれています．各レジスタは，32ビット長です．大半の状況では，どちらの種類のレジスタでも使用できます．ただし，特定の種類のレジスタを必要とする場合もあります．どのレジスタでも，**ワード**（16ビット）と**ロングワード**（32ビット）のデータ量に対する処理に使用することができ，主記憶のインデックス付きアドレッシング（第2章を参照）にも使用することができます．**バイト**（8ビット）のオペランドに対しては，データ・レジスタしか使用できません．スタック・ポインタまたは主記憶のアドレッシング用のベース・レジスタとしては，アドレス・レジスタしか使用できません．レジスタA7は重複しています．どちらの物理的レジスタが実際に使用されるかは，プロセッサがスーパーバイザモード（後述）かどうかによって決まります．

主記憶は多数のバイトから成る記憶領域によって構成され，何バイトあるかは個々のコンピュータ・システムにより違います．各バイトには**アドレス**という識別番号があり，メモリは通常（ただし必ずしもそうでない場合もあるが），各バイトがアドレス0，1，2，…，N－2，N－1（N＝メモリ内の総バイト数）という順序で並んでいます．直接アクセス可能なメモリのサイズは，非常に大きく，最高16,777,216バイトです．68000はバイト，ワードまたはロングワードの各メモリ単位に対して処理を行うことができます．ワードは2個の連続するバイトで，最初のバイトが偶数アドレスになっています．ロングワードは4個の連続するバイトで，やはり偶数アドレスから始まります．ワードまたはロングワードのアドレスは，先頭（最も低い番号の）バイトの（偶数）アドレスです．

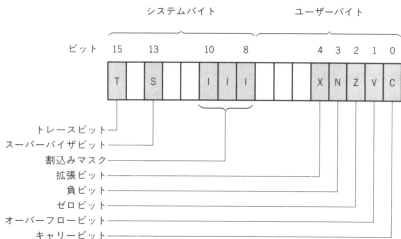
ここで注意しておくべきことは，68000のアドレスが常に24ビットで表されるという点です．したがってアドレスをロングワード，またはレジスタに収容した場合，8個のスペアビットが生じます．これはアドレスが常に正の数であることを意味し，よって2つのアドレスのうちどちらが上かを比較する場合，混乱が生じません．

16ビット・ワードおよび16ビット・アドレスを持つ他のコンピュータで，負のアドレスを使用する場合がありますが，これは非常に解析しにくいエラーの原因になる恐れがあります．状況によっては8個のスペアビットを利用するのが非常に便利な場合があり，ロングワードのポインタとともに，何らかの追加情報を収容することができます．この追加情報の使途としては，このポインタがどんなオブジェクトを参照するかを示したり，または単に，これが値それ自体ではなく，値に対するポインタであることを示すフラグとして使用することができます．ここで注意しておくべきことがあります．モトローラ社の暫定情報によると，68000シリーズの将来的なモデルでは，完全な32ビット・アドレスを使用する予定のため，スペアの8ビットを利用した場合，プログラムを新しいモデルへ移行するのが困難になるということです．

主記憶は，コンピュータが処理するデータを収容する他に，コンピュータにどんな処理を行うかを指示する命令も収容しています．各命令は，1～5ワードを占有し，1個のオペレーション・ワードと，0～4個のオペランド・ワードから構成されています．オペレーション・ワードは，どんな動作を実行するべきか(そして，暗黙的にその命令全体で何ワードあるか)を指定します．オペランド・ワードは処理するべきデータがレジスタまたは主記憶のどの位置にあるか，そして結果をどこに収容すべきかを示します．

命令は通常，メモリ中に置かれた順序にしたがって，一度に1つずつ実行されていきます．これは，料理の本に書かれている順序で料理をしたり，楽譜どおりに演奏するのと似ています．

プログラム・カウンタと呼ばれる特別なレジスタがあり，それは次に実行する命令のアドレスを収容するために使用されます．いくつかの命令，例えば**ジャンプ**(飛越し)または，**分岐**と呼ばれる命令は，実行順序を変更する作用を持ち，特定のアドレスにある命令へ強制的に制御を移します．この機能によって，コンピュータが1つの動作を反復実行したり，あるいはデータの値によって異なる動作を行うというような処理が可能になります．

ステータス・レジスタと呼ばれるもう1つの特殊なレジスタがあります．これは，コンピュータの状態に関する特定の事項を記憶するために使用されます．ステータス・レジスタの構成は次のようになっています．

システムバイトのビットについては，第7章で詳しく説明します．

○**トレースビット**…………プロセッサがトレースモードの場合に1，それ以外の場合は0にセットされる．
○**スーパーバイザビット**…プロセッサがスーパーバイザモードの場合に1，ユーザーモードの場合に0にセットされる．
○**割込みマスク**……………7個の割込みレベルのうち，どれが使用可能になっているかを示す．

ユーザーバイトには5種類のコンディション・コード・フラグが入ります．これらのフラグは，算術または比較演算など，特定の命令によってセットされ，あとの段階の命令に対し，演算結果に関する情報を伝えます．コンディション・コードの意味は次のとおりです．

Z：結果はゼロ
N：結果は負の値
V：2の補数演算中にオーバーフローが発生(すなわち，結果が大きすぎてデスティネーションに入りきらない)

C: 桁上がり(または，減算の場合の桁下がり)の発生

X: 拡張フラグ．これは多倍長操作(例：2個の64ビット数の加算)で使用される．拡張フラグは桁上がりフラグと同様にセットされるが，Cよりも X を変更させる命令の方が少ない．

コンディション・コードの設定状態は，Bcc, DBcc および Scc の一連の命令(第3章で説明)によりテストすることができます．

68000の命令は少数のグループに自然に分けられ，第2章以降では，それぞれ1つのグループごとに説明していきます．多くの命令はデータの移動(メモリ間の移動，レジスタ間の移動，またはレジスタとメモリ間の移動)に関連しています．その他の命令は，データの加算や比較など，算術演算や論理演算を行い，分岐とジャンプはプログラムの流れを制御します．またプロセッサを停止したり，コンピュータに接続されている外部装置を制御したりするさまざまな命令もあります．

1.3 位置独立コード

コンピュータのプログラムは多くの場合，データやジャンプの飛び先がどこであるかを指定する，固定的なメモリ・アドレスを含んだ状態で書かれています．このようなプログラムはメモリ内の特定の場所にロードする必要があり，そうでないと作動しません．これは一度に実行するプログラムがただ1つしかないような，単純なコンピュータ・システムでは問題ありません．しかしプログラムをメモリ内の任意の場所へ置くことができればずっと便利です[注4]．このようなプログラムのことを**位置独立コード**(position independent code)で書かれたプログラムと言います．

68000の命令セットでは，記憶域の任意の場所へロードすることができるプログラムを簡単に書くことができます．その理由は，プログラムのジャンプを生じさせる命令が，ジャンプの飛び先を**アブソリュート**(絶対的)ではなく，**リラティブ**(相対的)に指定できるからです．例えば，分岐を指定する形式は，〝アドレス5000にある命令に行く〟というような形式である必要はなく，〝この命令の

192バイト前の命令に行く”という形式を使うことができます．後者の形式を用いると，プログラムがメモリ内のどの場所にあってもいいわけです．このような分岐命令はほとんどのコンピュータにありますが，普通，現在の命令から128バイト先までしかジャンプできず，これでは不便を生じる場合が多くあります．その点68000は，最高32768バイトまでジャンプすることができ，どんなプログラムでもこれで充分だと思われます．

位置独立コードのもう1つの側面は，データのアドレッシングに関連しています．68000の持つ豊富なアドレッシングモード(第2章を参照)により，レジスタが保持しているアドレスに対して相対的にデータをアクセスすることができ，そのためプログラムは，メモリ内の任意の位置にデータ領域を簡単にセットすることができます．

真の意味で位置独立なプログラムというのは，初期的にはメモリ内の任意の位置に置くことができ，その後，実行中に他の位置に移動することができるプログラムのことでしょう．このようなプログラムは，プログラム・カウンタ相対でデータをアドレスする必要があります．68000ではプログラム・カウンタに相対する位置からデータを読むことはできますが，この方法でアドレスされたデータを変更することはできません．これはよくできた機能であり，プログラミングで重要とされる，プログラム領域とデータ領域の明確な分割を目指したものです[注5]．

以上のように68000では，メモリ内の任意の場所にロードでき，データ領域を任意のところにセットできるプログラムを素直に書くことができます．また，データ領域が同じ場所にある場合，実行中に移動できるプログラムを容易に書くこともできます．

1.4　チップが提供するデバッグ支援機能

68000には，プログラミングエラーを容易に検出し，その番地を指定できる機能が数多く備わっています．このような機能の中には，不正動作に対する組込みのチェック機能や，プログラマがプログラムをデバッグする際の支援手段として使用できるものもあります．

プロセッサには，特定の状況が発生した場合に，強制的にハードウェア・トラップを行う機能があります．すなわち，命令実行の通常の流れに割込みがかけられ，実行の停止した位置を記録し，メモリ内の固定的な位置へジャンプします．この固定的な位置には，適切な処置をとるプログラムが置かれている必要があります．例えば，ユーザーに対して何が発生したのかを知らせ，ユーザーがプログラムを実行し続けるのかどうかを質問するエラーメッセージを表示するなどです．もしユーザーがプログラムの続行を希望するならば，実行が停止した位置にジャンプします．

次のような原因によってトラップが発生します．

○奇数アドレスによるワードまたはロングワードのアクセス
○未実装命令や不正命令の使用
○存在しないメモリへのアクセス
○ゼロによる除算
○周辺装置からの擬似割込み

ある種の命令もまた，トラップの原因となります．TRAPV 命令は，直前の算術演算でオーバーフローが発生した場合にトラップを発生させます．プログラム内にこのチェック機能が必要な場合は，各算術演算の後にこの命令を指定します．同様に，CHK 命令は，レジスタ内の値が指定された数よりも大きい場合にトラップします．この命令は，メモリ・アクセスが一定のデータ領域内に対して行われているかどうかをチェックする目的で使用することができます．

TRAP 命令は，常にトラップを発生させます．この命令をプログラム内の重要なポイントに使用して，プログラムを停止させ，レジスタやメモリの内容を検査することができます．こうしてプログラムの動作を段階ごとにチェックすることができます．

最終的なデバッグ手段は，プログラムの個々の命令を一度に1つずつ実行する方法であり，どこで障害が発生したかを正確に検出できる，非常に強力な方法と言えます．この操作を行うには，ステータス・レジスタ内の1つのビットをセットし，マシンをトレースモードにします．このモードでは，各命令が処理された後に，トラップが行われます．適切なデバッギング・プログラムを使

ってこのトラップをつかまえることにより，ユーザーはプログラムの重要な部分を一段階ずつ調べて，プログラムの動作を詳細にチェックすることができます．これは，トレースモードのないコンピュータでは実現するのが非常に難しい機能です．

デバッグでのトラップの使用法は，第7章で詳しく説明します．

1.5 高級言語のサポート

68000はそのユーザーの多くがアセンブリ・コードではなく高級言語(例：FORTRAN, Pascal, または Algol68)でのプログラミングを望んでいることを意識して設計されました．高級言語で書かれたプログラムは，アセンブリ言語と比べて，普通の英語にはるかに近い形式になっています．つまり，アセンブリ・コードよりも短時間で，しかも簡単にプログラムを書けることを意味し，一般的に誤りも見つけやすくなります．このようなプログラムは，移植しやすいという利点もあります．つまり，その言語を使用できるマシンなら，どんなマシンでも走らせることができます．これに対し，アセンブリ言語のプログラムは，それを作成したコンピュータと同じ機種でしか実行できません．

コンパイラと呼ばれるプログラムは，高級言語で書かれた文を機械語(コンピュータが理解できる命令)に変換します．コンパイラによって作成される機械語は，人間が書くコードに比べると一般的に冗長です．ある動作を行うのに，実際に必要とされるよりも多くの(つまり無駄な)命令を使う傾向があり，そのためコンパイラが出力するコードは，人間が書いたコードよりも大きなメモリを占有し，実行速度も遅くなります．しかし，演算能力とメモリが比較的安価であれば，人間の労力を使うよりも，コンパイラを使った方がはるかに都合が良いと言えます．

68000の特長として，高級言語に対するコンパイラの作成が簡単であり，しかも効率の良いコードが作成できる，という点があります．16個の汎用レジスタを装備しているため，頻繁に使用するポインタや値を常にレジスタに入れておくことができ，単にメモリとレジスタの間で値をやりとりするだけのコードは，非常に少なくてすみます．命令とアドレスモードが規則正しく一貫性のある構

造をとっていることにより，実際の機械語の生成に関連するコンパイラの部分が単純になります．大半の命令は，3種類の異なるサイズのオブジェクトに対して使うことができ，いずれのアドレスモードでも実行可能です．広いメモリ領域を直接アドレス指定できる機能により，言語に対する記憶域の編成が簡素になります．

高級言語専用の命令が，いくつか準備されています．高級言語のプログラムは通常，独立したモジュールやルーチンが結合されて1つの完全なプログラムを作成するようになっています．コンパイラがあるモジュールをコンパイルしているとき，プログラム内のどの部分からそのモジュールが使われているのかを，コンパイラは認識していません．したがって，そのモジュール内ではどのレジスタおよびメモリのどの領域を使用するのが安全かということも，認識していません．どのレジスタを使用できるかという問題を解決する最も簡単な方法は，モジュールに入るときにいくつかのレジスタの内容を保存しておき，モジュールから出るときに，その内容をすべて復元することです．この処理を行うのがMOVEM命令です．この命令は，指定した一連のレジスタの内容をメモリにコピーし，再び同じレジスタにコピーします．任意のグループのレジスタを保存または復元できるので非常に融通性があります．

LINKおよびUNLK命令により，呼び出された個々のプログラム・モジュールが，スタック(第4章を参照)上に局所的な記憶域を割り付けることが可能となります．LINK命令はフレーム・ポインタをスタックに待避させ，指定した大きさの領域をあらたにスタック上に確保する働きをします．UNLK命令はこの逆の処理で，割り当てられた領域を解放し，ポインタを元のポインタへ復元します．

デバッグの補助手段の項で説明されている命令(例：CHK, TRAPV)は，コンパイル済みのコードでも活用することができ，算術演算時のオーバーフローや添字の誤った使い方などによる異常なデータ・アクセスを即座に検出することができます．このような検査命令は単一の命令なので，プログラムに埋め込んでも，実行速度の低下は微々たるものです．人間が同じことをするのは困難ですが，コンパイラは，このような命令を適切な位置に確実に置くことができます．

1.6 オペレーティング・システムのサポート

コンピュータそのものは，どちらかというと扱いが難しいものです．コンピュータにできるのは，ただ単にそのコンピュータ自身の機械語でコーディングされた命令を実行するだけです．このような理由から，通常，コンピュータを使いやすくするためのプログラムが開発されています．このようなプログラムのことをオペレーティング・システムといい，非常に単純なオペレーティング・システムのことをモニタと呼ぶ場合があります．

典型的なオペレーティング・システムは，コンピュータに接続されたすべての周辺装置を制御し，ユーザーが端末装置から打ち込んだコマンドを解釈し，人間にとってわかりやすいファイル名を付けてディスク記憶域を管理します．また，いくつかのプログラムを，見かけ上同時に実行することができる機能もあります(実際には，短い間隔でプログラムを切り換えている)．さらにプログラムの実行中に生じたエラーを処理し，ユーザーに対して適切なメッセージを表示します．またメモリ中のユーザー・プログラムやレジスタの内容を，ユーザーが検査するためのコマンドも準備しています．

68000には，オペレーティング・システムのサポートに必要でしかも便利な機能が数多く備わっています．これにより，オペレーティング・システムは，ユーザー・プログラムの実行によって引き起こされる損傷から保護され，プログラムに対する制御を保つことができます．これを実現するためには，2種類のプロセッサモード，つまりスーパーバイザモードとユーザーモードを使用します．オペレーティング・システムは，スーパーバイザモードで作動し，他のプログラムを実行する前に，ユーザーモードに切り換えます．いくつかの重要な命令は特権化されており，ユーザーモードでは実行できません．プロセッサチップは，メモリまたは周辺装置へのアクセス中のモードを示す出力信号線を備えているので，周辺装置とメモリの一定の領域がスーパーバイザモードだけで使用可能となるようにハードウェアを接続することができます．このようにして，オペレーティング・システムは，そのコードと専用の作業領域を収容した記憶域を保護することができ，周辺装置にアクセスできる唯一のプログラムであることが保証されます．このためには，ユーザー・プログラムが絶対にスー

パーバイザモードにセットできないことを保証し，かつユーザー・プログラムでオペレーティング・システム・ルーチンを呼び出して，そのルーチンをスーパーバイザモードで実行させることができなければなりません．前者は，モード変更命令が特権命令であることから保証されます．後者は，TRAP 命令(第7章を参照)を使って，ジャンプとモード変更を同時に行うことにより，実現されます．

68000には，ベクタ付き割込みとトラップ(第7章を参照)があります．これは，個々の周辺装置がプロセッサに対して信号を出し，その装置を取り扱うための適切なコードへ直接ジャンプできるようにするものです．したがって，オペレーティング・システムが簡素になります．また複数の割込みレベルが準備されているため，最も緊急を要する割込み信号を最初に処理できるように，個々の装置からの信号を編成することができます．

MOVEM 命令はオペレーティング・システムでも便利な命令です．割込み，またはトラップが発生すると，オペレーティング・システム内のどこかへ即座にジャンプします．割込みハンドラは環境を保存するために必要なレジスタの退避/回復を MOVEM 命令で効率良く行えます．もう一つの特殊な命令は MOVEP です．この命令は特に周辺装置へのデータ移動を簡略化するための命令です．

2つのプログラムを並行して実行する場合，どちらか一方が，何らかの資源(resource, 例：周辺装置，記憶装置など)に対して排他的アクセスできるようにする必要がしばしばあります．最も簡単な方法は，資源が利用可かどうかを示すフラグ・バイトをメモリ内に持つことです．フラグを調べ，資源が解放されるまで待ち，解放されたらフラグを占有状態にセットします．ただし，フラグを検査してセットする動作は不可分な操作として実行されなければなりません．さもないと2つのプログラムが双方ともフラグが利用可状態であるのを見て，同じ資源を要求する場合が考えられるからです．TAS(テスト・アンド・セット)命令が，この目的のために用意されています．さらにこの命令は，同一のメモリを共有する，いくつかのプロセッサで実行されるプログラム間のインターロックにも使用することができます．というのは，プロセッサは TAS 命令の実行中ずっとメモリの制御を保持するからです．このことを**リード・モディファイ・ライト・サイクル**と呼ぶ場合があります．

68000にはバス調停(bus arbitration)回路があり，バスに接続されているすべ

ての装置がバスを共有できます．これには，記憶装置，端末装置，ディスクおよび他のプロセッサが含まれます．インテリジェント・デバイスは，プロセッサに割り込む必要なしに直接メモリにアクセスすることができます．このような動作を**ダイレクト・メモリ・アクセス**(DMA)といいます．例えば，プロセッサはディスク装置に対し，ディスクからメモリへデータを転送するよう要求することができます．ディスク装置はDMAによって転送を行い，終了したときだけ割込みを行うため，プロセッサはそれまでの間，他の処理を実行することができます．

1.7 典型的なアプリケーション

68000の実行速度や大容量のメモリを事実上必要としない環境で使用するには，68000システムは高価すぎるというのが現状です．そのようなアプリケーションは，安価な8ビット・マイクロプロセッサの守備範囲であると言えます．68000は，コンピュータの端末装置，グラフィック・ワークステーション，ワードプロセッサ，医療機器など高性能が要求される環境においてこそ適しています．68000は汎用コンピュータとして考えたとき，あらゆる大きさのミニコンピュータの強力なライバルとなっています．大きなアドレス空間を持っているということは，以前はメインフレームでしか実行できなかったプログラムを実行できる強力なパーソナル・コンピュータになり得る，ということを意味します．さらに，複数のユーザーを同時にサポートする目的でも使用できます．ただし，この場合，各々のユーザーを分離するために，何らかのメモリ・マッピング用ハードウェアが必要です．

1.8 68000シリーズのプロセッサ

68000シリーズのプロセッサの中で，68000は一番最初のモデルです．本節では，本書執筆の時点で発表されている68000シリーズの他の3つのモデル（68008, 68010, 68020）について簡単に説明していきます[注6]．

68008は，68000に８ビット(16ビットではない)の外部データバスを付けたものです．これによりプロセッサは，８ビット・サポートチップとともに使用できます．その結果，実行速度は犠牲になりますが，回路の複雑性とコストはいくらか改善されます．

68010は68000ときわめて類似していますが，オペレーティング・システム・サポートの改善と高速化のためいくらか変更が加えられています．VBR(ベクタ・ベース・レジスタ)という新しいレジスタがあり，割込みベクタ(第７章を参照)のベース・アドレスがここに入ります．このレジスタはリセット後の状態では０にセット(68000との互換性のため)されていますが，変更可能とすることにより，異なるオペレーティング・システムのプロセスが自分自身のトラップを直接的な方法で処理できます[注7]．

例外処理後，スタックに記憶される情報に対して，さまざまな変更が加えられています．特にバスエラー(アドレスエラー)の原因となった命令が継続可能となります．これによって，仮想メモリ付きのシステムが可能となり，物理的に使用可能なメモリより大きいメモリを，プログラムがアクセスできるようになります．オペレーティング・システムは，仮想メモリの実際に使用されているセクションがいつでも実記憶上にあり，残りのセクションが，ディスクなどの２次記憶上に収容されていることを保証します．仮想アドレスから実アドレスへの変換は，実記憶上にない番地を使用するとバスエラーが発生する，という方式で行われます．オペレーティング・システムは，仮想アドレス空間の関連する部分を実記憶に読み込み，次にバスエラーの原因になった命令の実行を継続することによってバスエラーに応答します．

68010には，MOVECとMOVESという２つの新しい命令があります[注8]．MOVECは，各種の制御レジスタ(VBRも含む)へのアクセスに使用されます．MOVESは，通常はアクセス不可能なアドレス空間の読み書きを可能にします．データのアクセスは，通常，現在の特権レベルにしたがって，ユーザーデータまたはスーパーバイザ・データ・アドレス空間に対して行われます．ただし，MOVECによってセットされる３ビット・ファンクションコード・レジスタは２個あるので(１個はソース用，もう１個はデスティネーション用)，スーパーバイザモードで作動しているプログラムは，MOVESを使って，スーパーバイザ・プログラム，ユーザー・プログラムまたはユーザー・データ・アドレス空間内のある

場所を読み書きすることができます．

68010では32ビットの算術および論理演算，CLR, Scc, および MOVE SR も含めてさまざまな命令を68000よりも速く実行することができます注9．また，バスエラーのタイミングも緩和されているため，エラー検出機能を持つメモリ・システムにおいても，実行速度が犠牲になりません．

68020プロセッサは，68010のすべての新しい機能に加えて32ビット演算のサポートを強化しています．68020は完全な32ビット外部データバス，分岐命令における32ビット・オフセット，およびインデックス付きアドレッシングモードにおける32ビット変位が含まれます．CHK, LINK, MUL, および DIV の各命令は，32ビットのオペランドをとることができます．追加的なアドレッシングモードが使用可能であり，2レベルの間接アドレスを使ったインデックス付きアドレッシングが可能です．

68020には命令キャッシュがあり，命令をメモリから繰り返し取り出す必要がないので，小型のループを非常に速く実行することができます．さらに，完全なコ・プロセッサ・インターフェイスを装備しており，他のチップ(例：浮動小数点演算プロセッサ)を追加することにより，命令セットを拡張することができます．

68020で使用可能な新しい命令がいくつかあります．これらの命令には，高級言語のプロシージャ・コールのためのより洗練されたエントリーおよびエグジット操作(CALLM, RTM)，そして，さまざまなサイズのビット・フィールドを扱うための命令が含まれます．パックされた10進数データに対する命令の範囲は，PACK，および UNPK によって拡張され，文字と10進数の間の変換を行います．

監訳者注

注1： 「M68000マイクロプロセッサ　ユーザーズマニュアル」CQ出版
(1984年10月発行)

注2： コード,データ,スタックの3種類がユーザー，およびスーパーバイザモードで使用できる.

注3： 一般にはZ8001単体ではなくメモリ・マネージメントユニット(MMU,例えばZ8010)とともに使用したときに，8Mバイトまでアクセスできる.

注4： 例えば，異なるハードウェア構成のシステムに対しても，同じコードが少ない変更で利用できるとか，同時に複数のプログラムが走るシステムでも，コードの書換えをせずに適当な番地にロードすることができる.

注5： もし，これが変更されると困ったことが生じる，典型的な例を次に挙げると，

- ●自分自身を書き換えるコードは共有することができなくなる．もちろん，リエントラントではなくなる.
- ●68000では，ステータス信号(FC0～FC2)により，メモリアクセスがプログラムコードかまたはデータに対するものなのかを識別することが可能であり，メモリ保護を行うシステムではこれを利用している．例えば，プログラムは読み出し専用で変更不可とし，データは書換え可能として保護を行うとすると，プログラムコードを変更するプログラムは正しく走らなくなってしまう.

このため，プログラム領域とデータ領域の分割は望ましいものであり，プログラム領域は各プロセス(プログラムとデータが組となった，いわゆるジョブ)で共有され，データ領域は各プロセスごとに別々に割り当てるのが一般的である.

注6： この他に68010の拡張版で68012があり，アドレスが31ビットになっている.

注7： 各オペレーティング・システムはそれぞれ固有のVBRの値を持ち，核となるオペレーティング・システムがその切換えを行う.

注8： この他にRTD，MOVE from CCRが追加されている．RTD命令ではRTS命令の動作に加えて，スタック領域を指定した大きさだけ解放する．これは高級言語で手続きからの戻りで使われる．MOVE from CCRはMOVE from SR命令が特権化されたため設けられたものである.

注９：　Loopモードが新設されている．例えば，文字列に対し，

```
LOOP     MOVE.B  (Ax)+,(Ay)+
         DBRA    Dn,LOOP
```

というコードを実行すると，第１回目のループでは命令のフェッチを行うが，２回目からはフェッチを行わなくなり，ループ内ではオペランドアクセスだけが行われ，従来の68000と比較して高速にループを実行できる．

アセンブラの構文とアドレッシングモード

2.1 アセンブラの構文 38
2.2 アセンブラ・ディレクティブ 41
2.3 アセンブラの構文のまとめ 45
2.4 式 46
2.5 アドレッシングモード 48
2.6 実効アドレスの分類 61

はじめに

本章では，第3章以降で説明するさまざまな命令を理解する上で必要な予備知識を述べます．内容としては，アセンブラの構文(プログラムの記述方法)とアドレッシングモード(命令が作用するデータを指定する各種の方法)についてです．

2.1 アセンブラの構文

コンピュータ自身が理解する唯一の言語は**機械語**です．機械語はメモリ内の単なるビット・パターン，または数の並びとして考えることができます．機械語形式のプログラムは，人間にとっては理解または記述するのがどちらかといえば難しいものです．そのため，機械語に直接対応する**アセンブリ言語**でプログラムを書く場合が多いのですが，その場合には，命令，およびレジスタを示すのにニーモニック名を利用します．さらに，プログラム内のアドレス，およびその他の値を示す上で記号名も使用できます．**アセンブラ**というプログラムは，アセンブリ言語から機械語への変換に使用されます．以下に説明するアセンブリ言語の形式は，モトローラが使用する形式と同じであり，モトローラのアセンブラによって受け付けられます．これと異なるアセンブラを使用している場合，いわゆる方言(異なる記法を必要とする)を使用しなければならないこともあります．具体的な違いについては，各々のマニュアルを参照してください．

プログラムは，命令と呼ばれる一連のステップから構成されています．個々の命令は，1行のアセンブリ言語として書かれています．命令それ自体は，3，4，ないし5文字のニーモニック名を持っており，命令によってはその名前だけを行として書けばいいものもあります．一例を次に示します．

```
NOP
```

これはまったく何もしない命令です(このような命令がまったく無意味というわけではありません．デバッグのとき，望ましくない命令を置き換える目的で使うと便利ですし，また非常に短い遅延が必要な場合にも，使うことができます)．

命令の名前が，第1文字目から書かれている点に注意してください．その理由は後で明らかになります．

ただし，大半の命令では，名前だけでは不充分です．その命令の作用するデータが，レジスタまたはメモリの何処にあるのか，ということも指定しなければなりません．このために名前の後に(1個ないし複数の空白を開けて)オペランドを付けます．例を示します．

```
        CLR     D3
```

これは，データ・レジスタ3の下位16ビットをゼロにクリアします．オペランドを2個使用する場合には，コンマで区切ります(空白は入れません)．通常，左側のオペランドは，値が読み出されるべき**ソース**であり，右側のオペランドは，結果を置くべき**デスティネーション**です．オペランドをこの順序で書くという点に注意してください．特に他の表現で記述するコンピュータ用のアセンブリ言語に慣れている人は，注意が必要です．簡単な例を示します．

```
        MOVE    D1,D4
```

これは，データ・レジスタ1の下位16ビットをレジスタ4に単にコピーします(両方のレジスタとも，残りの部分には影響を与えません)．

68000には，多くの命令が，3種類の異なるデータ・サイズに対して作用できるという便利な特長があります．それらは，**バイト**(8ビット)，**ワード**(16ビット)，**ロングワード**(32ビット)です．どの長さが必要であるかを示すために，名前に“**.B**”，“**.W**”，または“**.L**”という**サイズ指示子**(サフィックス)を付けます．サイズ指示子をまったく付けない場合は，“**.W**”であると見なされます．したがって，上の命令は，次のように書くのと同じです．

```
        MOVE.W   D1,D4
```

レジスタの32ビットすべてをコピーするには次のように書きます.

```
        MOVE.L   D1,D4
```

同様に, レジスタの下位8ビットだけをクリアするためには, 次のように書きます.

```
        CLR.B    D3
```

長さを指定するサイズ指示子を常に使用する(すなわちオプションの".W"を省略しない)習慣をつけるのは良い考えです. というのは, 誤って命令のワード形式を使うのが, 68000でよく起こるプログラミングエラーだからです. この種のエラーは, 追跡するのが難しく, あいまいな障害をプログラム内で発生させる原因となります.

命令とそのオペランド(もしあれば)の後に何かがあっても, アセンブラによって無視されます. そのため, プログラムにコメントを入れて, 人間が読んでも理解しやすくすることができます. また, 行の先頭にアスタリスク"*"がある場合, その行全体はコメントとして取り扱われます.

```
* This whole line is a comment
        CLR.L    D3        A comment after an instruction
```

プログラムに積極的にコメントを入れることをお勧めします. プログラムの作成中にコメントを入れていくのは, 面倒に思えるかもしれませんが, プログラムの作成者以外の人がプログラムを理解するのにたいへん役に立ちますし, また, プログラムを書いた後で作成者自身が修正するときにも役立ちます.

上記の各例で, 命令名は左マージンに合わせて書いてあります. 行の先頭が空白ではない場合, 最初の項目はラベルと見なされます. ラベルとは, その行にある命令のメモリ・アドレスを示す記号名です. ラベルの名前は, 英字で始まり, 英字と数字だけを含む任意の語です(実際には, ほとんどのアセンブラで

これら以外の文字を使用することができます．本書の例では，名前を読みやすくする目的で，アンダースコア〝_〞を使用します)．アセンブラはラベルと，そのラベルが参照するアドレスを記憶します．そして，プログラム内の他の部分でラベルを使って，そのアドレスを参照することができます．この特性が特に便利なのはジャンプ命令で，指定したアドレスで実行を継続することができます．

```
CLRD3    CLR.L    D3             Labelled instruction
*          :
*          :
         JMP      CLRD3          Jump to instruction
*                                labelled CLRD3
```

また，ラベル名の後にコロンを付けることによって，ラベルをマージンから段付け(indent)することもできます．

```
CLRD3: CLR.L   D3
```

このようにラベルを利用することにより，ユーザーはこの **CLR** 命令の実アドレスを知る必要がなくなり，またプログラムの残りの部分に対する変更のためにこのアドレスが変化する場合でも **JMP** 命令を変更する必要がありません．

2.2 アセンブラ・ディレクティブ

アセンブラは，命令とコメントの他に，アセンブラ自身に対するコマンドであるディレクティブを受け入れます．ディレクティブは命令と同じ方法で書きますが，ただし(**DC** および **DS** は例外として)コードは生成されません．ここで説明するディレクティブは，大半の68000アセンブラで同形式で使用可能と思われる，いくつかの基本的なものです．大半のアセンブラには，これ以外のディレクティブがあり，アセンブリ・リスティングでのレイアウトや，生成されるオブジェクト・モジュールの形式を制御したり，条件付きアセンブリおよびマクロの機能を提供します．

ディレクティブの1つの例として**EQU**があります．このディレクティブは，記号名に対して値を定義します(アドレスに対する名前であるラベルと類似しています)．

```
SIZE    EQU     100
```

これは，SIZEを値100の名前とします．プログラム内で〝SIZE〟を使用すると，アセンブラはあたかも〝100〟がそのかわりに書かれているものとして動作します．これは，いくつかの点で便利です．プログラムの複数の個所でこの値を使用する場合，〝100〟と個々の箇所で明示的に書くよりも，始めにSIZEを定義しておき，それをプログラム全体で使用した方が，はるかに変更が簡単です．また，数のかわりにニーモニック名を使用した方が，人間にとってわかりやすいプログラムになります．

68000のメモリは8ビット・**バイト**の配列として考えられ，上位アドレスに向かって0，1，2，…と番号が付いています．メモリバイトの番号は，そのバイトの**アドレス**と呼ばれます．アセンブルされたコードのメモリ内の位置を制御するディレクティブは2つあります．1つはORGで，**オリジン**(すなわち，最初の命令)として特定のアドレスを指定します．

```
        ORG     1024
START   CLR.L   D3
*        :
*        :
```

このシーケンスは，アセンブラに対し，コードが1024番地以降に置かれるものと想定してコードを生成させます．したがって，ラベルSTARTの値は1024となり，アセンブルされたコードが正しい位置にロードされるようこのアドレスで印付けされます．ORGで始まるコードは，アドレスが固定されているため，**アブソリュート・コード**と呼ばれます．その中にあるラベルは，アブソリュート・シンボルであるといいます．ORGを含むプログラムは，特定のアドレスに対する明示的な参照を含んでいるため，位置独立的ではない傾向があります．ORGのもう1つの形式として，ORG.Lがあり，これはアブソリュート・アドレッシングモード(以下の説明を参照)のアセンブリに影響を与えます．

プログラム中の数を10進表記よりも16進表記(16を基数とする)で書いた方が便利な場合が多くあります．使用する数字は，0～9と，A～F(10～15を表す)です．アセンブラは，ドル記号で始まる16進数を受け付けます．

```
        ORG     $400
```

これは，ORG 1024 (＝4×256＋0×16＋0×1)と同じです．本書全体を通じて，16進数を表すのに〝$〟を使います．

ORGに対する相補的なディレクティブはRORG[注1]であり，プログラムが**リロケータブル**である(すなわち，メモリ内の任意の位置に置くことができる)ことを示します．RORGも引数をとりますが，この引数は，通常ゼロでなければなりません．上のプログラム部分を次のように変えます．

```
        RORG    0
START   CLR.L   D3
*         :
*         :
```

アセンブラは，STARTの値を未定なものとします．STARTは0という値を与えられ(なぜなら，セクションの先頭からのオフセットが0であるため)，そして，STARTがリロケータブルであるということが記録されます．STARTのようなラベルのことを，**リロケータブル・シンボル**(プログラムがメモリにロードされるまで値が未定なシンボル)といいます．リロケータブル・シンボルを使用した場合，アセンブラはつねに位置独立なコードを生成しようとします．

```
        RORG    0
START   CLR.L   D3
*         :
*         :
        JMP     START
```

JMP命令は，〝ここからXバイト前の命令へジャンプする(Xはアセンブラによって計算される)〟としてコーディングされます．位置独立コードが生成できない方法でリロケータブル値が使用された場合(例えば，この例でXの値が符号付き16ビット整数で表せないとき)，アセンブラは，コードの中にこれらのワー

ドのリストを含め，そのリストに示された値は，プログラムが実際に記憶域に置かれるときに決定されなければなりません．その時点まで，これらの値を知ることはできません．このリストを**リロケーション情報**といいます．

メモリ領域の予約と初期設定のために，2つのディレクティブが用意されています．DS(Define Storage，記憶域の定義)ディレクティブは，メモリ領域を予約するために使用します．このディレクティブは，割付け単位のサイズを示すサイズ指示子と，そのような割付け単位をいくつ予約するかを示すオペランドを1個取ります．次に例を示します．

```
BUFFER  DS.B    80          Reserve 80 bytes of memory
        DS.W    $20         Reserve 32 words
        DS      $20         Reserve 32 words
        DS.L    3           Reserve 3 long words
```

予約されたメモリは特定の値に初期設定されていません．サイズ指定子が，〝.B″でなければワード境界に領域が揃えられます．したがって〝DS.W　0″は，単にワード揃えを行わせるために使用することができます．DSディレクティブにラベルを付けた場合，そのラベルは，予約された最初の位置(揃えがあれば，その後)のアドレスを参照することになります．

DC(Define Constant，定数の定義)ディレクティブは，メモリ領域に特定の値をアセンブルするために使用します．このディレクティブでは，通常3種類のサイズ指定子，および1個ないし複数のオペランド(コンマで区切る)を取ります．

サイズ指定子が〝.B″でなければ，DSの場合と同様，ワード境界へ揃えられます．オペランドとしては，数，式，またはシングルクォーテーションで囲んだ文字列を使用することができます．

DCにつづく文字列は特殊な方法で取り扱われます．文字定数(2．4章を参照)と見なされるのではなく，各文字について1バイトがアセンブルされます．

DC.WまたはDC.Lを使用した場合，最終的なワードまたはロングワードには，必要に応じてNULL(0)が付加されます．

```
MESSAGE DC.B    'Hello'       5 bytes containing the
*                             codes for 'H', 'e', etc.
        DC.L    'Hello'       8 bytes are assembled
*                             the last 3 hold zeros.
        DC.W    10,20,30      3 words are assembled
        DC.L    $FF,99        2 long words are assembled
```

END ディレクティブは，単にアセンブラ・プログラムの終わりを示すために使います．どんなプログラムも最後の行は，次のようになっていなければなりません．

```
        END
```

○アセンブラ・ディレクティブのまとめ

ディレクティブ			機　能
〔ラベル〕	DC.s	式，式，…	サイズsの値をアセンブルする
〔ラベル〕	DS.s	n	サイズsの領域をn個確保する
	END		ソース・プログラムの終り
シンボル	EQU	値	シンボルを値に定義する
	ORG	アドレス	アブソリュート・セクションのオリジンを設定する
	RORG	アドレス	リロケータブル・セクションのオリジンを設定する

2.3 アセンブラの構文のまとめ

アセンブラの行は，コメント行，命令行，およびディレクティブ行の3種類に大別されます．コメント行はアスタリスク〝*〟で始まり，そのあとには任意の文字を使用することができます．

```
* This is a comment line
```

命令行の一般的な形式は，次のとおりです．

```
label    opcode   operand(s)    comment
```

各フィールドは，少なくとも1個の空白によって次のフィールドと区切られ，ラベル，オペコードおよびオペランド・フィールドの内部には，空白が入っていてはなりません(引用符付きの文字列の内部は例外)．ラベルおよびコメントは，必要に応じて使います．オペコード・フィールドは，命令名と，オプションのサイズ指示子(".B"，".W"，".L"，または".S")から構成されます．オペランドの個数は，命令のオペコードから決定されます．オペランドがまったく不要な場合は，アセンブラは，オペコード・フィールドの後にあるものをコメントとして取り扱います．オペランドが2個ある場合は，コンマで区切らなければなりません(ただし空白は入れない)．

ディレクティブ行の一般的な形式は，次のとおりです．

```
label    directive argument(s) comment
```

ディレクティブによっては，ラベル・フィールドを使用できないものもあり，また逆に，必ず使用しなければならないものもあります．2個以上の引数がある場合は，コンマで区切らなければなりません．

2.4 式

前述したように，数を書くべき場所のほとんどに，その数を表す記号を書くことができます．実際に，記号と数字を含んだ算術式を，数のかわりに使うことができます．+，−，*(乗算)および/(除算)の，一連の算術演算子が用意されており，例えば次のように書くことができます．

```
DAYHRS  EQU     24            Hours in a day
DAYMINS EQU     DAYHRS*60     Minutes in a day
DAYSECS EQU     DAYMINS*60    Seconds in a day
```

整数演算を使って値が計算されるため，結果はすべて整数となります．ただし,除算の場合だけは問題があり，結果は小数点以下が切り捨てられます．したがって 7/3は2です．

数は10進数または16進数(前に〝$〟を付ける)で書くことができます．数を指定するもう1つの方法が文字定数です．文字定数は，1～4個の文字をシングルクォーテーションで囲んだもので，指定された文字を順に右側からつめ，余った左側にはゼロを付加した,ロングワードの値です(最右端が最下位バイトとなる)．文字定数は，単一の文字の場合に最も便利で，読みやすくするために文字に対する数値コードに替わって使用するべきです．次に例を示します．

```
CHARZ    EQU      'Z'         Code for letter Z
CASEDIFF EQU      'A'-'a'     Difference between codes
*                             for upper and lower case
*                             forms of same letter
```

前にも説明したとおり，記号にはアブソリュートとリロケータブルの2つの種類があります．アブソリュート記号は，数とまったく同じなので，演算については何の問題もありません．ただし，リロケータブル記号を使って有効な結果を得るためには，制約事項があります．基本的規則は，結果が元の記号に合わせて,アブソリュートまたはリロケータブルのどちらかでなければならない，ということです．したがって，リロケータブルな量を使った乗算または除算は認められません．また，2つのリロケータブルな値の加算も認められません．ただし，定数をリロケータブルな値に加減算することができ，結果はリロケータブルになります(結果は，同じリロケータブル･セクション内の異なる位置のアドレスです)．これを使って，次のように書くことができます．

```
         RORG     0           Relocatable section
START    CLR.L    D3          This instruction is two
*                             bytes long
         CLR.L    D4
*         :
*         :
         JMP      START+2     Jump to second CLR.L above
```

ただし，実際にジャンプしたい命令にラベルを付ける方が良い方法だと言えます．

アブソリュートな数からリロケータブルな数を減算するのは誤りですが，リロケータブルな数からリロケータブルな数を引くのは，完全に正しい方法です．その結果は，プログラム内の2つの番地間の距離を表すので，アブソリュートな数です．これは，距離を表しますから，メモリ内に置かれる場合はどこでも常に，同じ値になります．次にプログラム例を示します．

```
        RORG    0           Relocatable section
PSTART  MOVE.L  PEND-PSTART,D0 Set D0 to program length
*          :
*          :
PEND
```

最初の命令は，プログラム全体の長さ(単位：バイト)をD0にロードします．行自体にラベルを使用している点に注意してください．その値は，最後にアセンブルされたバイトのアドレスです．

○式のまとめ

式	結　果
リロケータブル＊任意の式	（誤り）
リロケータブル／任意の式	（誤り）
リロケータブル＋アブソリュート	リロケータブル
リロケータブル－アブソリュート	リロケータブル
リロケータブル－リロケータブル	アブソリュート
アブソリュート－リロケータブル	（誤り）

2.5 アドレッシングモード

68000のほとんどの命令は，さまざまな形式のオペランドを受け付けることができます．オペランドは，レジスタ内にあったり，各種の方法でアドレスされるメモリ上にあったり，または命令自体に含まれる場合もあります．命令セットは秩序立った方法で構成されているので，個々の命令とは無関係に各種のア

ドレッシングモードを説明することができます．任意のアドレッシングモードで(または，ほとんど任意のアドレッシングモードで)表現することのできるオペランドのことを，**実効アドレス**といいます．

○レジスタ直接アドレッシング

オペランド・データはデータ・レジスタの1つ，またはアドレス・レジスタの1つに収容されています．レジスタ名は，Dn または An(n は 0 ～ 7 の数字)という形式で書きます．

```
MOVE.L  A7,D5
```

これは，アドレス・レジスタ7の全32ビットを，データ・レジスタ5にコピーします．長さが〝ワード〟である場合，レジスタの下位16ビットだけが読まれるかまたは変更されます．サイズ指示子〝.B〟は，アドレス・レジスタに対しては使用できません．データ・レジスタの場合は，下位8ビットだけが影響を受けます．

○アブソリュート・アドレッシング

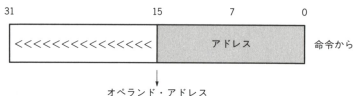
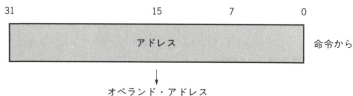
メモリ内のオペランドはその最初の(最上位の)バイトのアブソリュート・アドレスを与えることにより，位置指定することができます．オペランドは，単に数として書くか，またはその数を表すラベルか他の記号として書きます．1000(16進数)番地にあるバイトをクリアするには，次のように書きます．

```
CLR.B    $1000
```

アブソリュート・アドレスは，命令の中で16ビットまたは32ビットの数として表されるので，このアドレッシングモードには，実際には2つの形式があります．短い形式では，16ビットのアドレスは，使用される前に32ビットに**符号拡張**されます．つまり，16ビット数の最上位ビットがアドレスの上位16ビットとしてコピーされます．したがって短い方の形式は，メモリの最下位の32Kバイトと，最上位の32Kバイトまでの領域をアドレスするのに使用することがで

きますが，その間はアドレスできません．後方参照の場合，アセンブラは，参照される位置のアドレスをすでに知っているので，常に適切な長さを選択することができます．前方参照の場合に，このアドレッシングモードのどちらの形式を選択するかを制御するために，**ORG.L** は長い方の形式，**ORG** は短い方の形式を要求します．これは実際にどういう意味を持つのかというと次のようになります．すなわち，アブソリュート・コードのプログラムが32K(＝＄8000)を越える場合，**ORG.L** を使って，前方参照で16ビットを越える場合があることをアセンブラに対して知らせなければなりません．

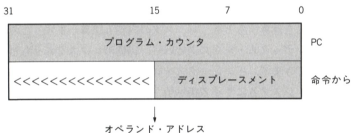
これらのアドレス計算を図で示します(″＜＜＜″は，符号拡張を表しています)．

アブソリュートモード(短い形式)

アブソリュートモード(長い形式)

○リラティブ・アドレッシング

プログラム・カウンタ(PC)の現在値と相対でメモリをアドレスする方法として，2つのモードがあります．これは主に，位置独立型コードにおけるジャンプのために使用されますが，プログラムに埋め込まれた定数を読むためにも使用することができます．この方法でアドレスされた位置に，書込みを行うことはできません．

これは**ピュア・コード**の作成を勧めるためのものです．ピュア・コードとは実行によってそれ自体を変化させないコードであり，そのため何回実行しても同じ作用をすることができます．また，同時に走るいくつかのプログラムの共有部分として実行することができます．ピュア・コードのことを，**リエントラント**である，と表現しますが，これは常に再使用可能であるという意味です．

〝ピュア〟より低級なコードとして，**順次再使用可能**(serially reusable)なコードというものがあります．これは，そのコードが終了したら，再び使用することはできるけれども，ただし，走っている間，常に適切な状態になっているとは限らないコードのことです．

2つのリラティブ・アドレッシングモードのうち，より単純な方のモードでは，メモリ・アドレスは現在のプログラム・カウンタと，16ビットのディスプレースメントの和として計算されます．命令の実行中，プログラム・カウンタの値は，命令の開始アドレスより2つだけ多い値となります．ディスプレースメントは符号付きの16ビット数として取り扱われるので，この方法で〔命令－32766〕~〔命令＋32769〕の範囲のアドレスを表現することができます．

オペランド・アドレスを，現在の命令の開始位置からのオフセット（これはディスプレースメント値となる）として書くことによって，このモードを要求することができます．現在の位置を参照するために，シンボルの〝*〟を使用することができます．したがって次のように書くことができます．

```
          JMP      *+10          Jump to the instruction 10
*                                bytes on
```

ただし，この方法はあまりお勧めできません．というのは，第一にオフセットをユーザー自身が計算しなければならず，第二に，中間に何らかの命令を挿入する場合，オフセットを忘れずに変更しなければならないからです．それゆえラベルを使った方が簡単で安全です．

リロケータブル・セクション内で，同じセクション内に定義されているリロケータブル・シンボルに対する参照が行われると，アセンブラは自動的にこのモードを生成します．次のように書いた場合，

```
          RORG     0
START     CLR.L    D3
*          :
*          :
          JMP      START
```

アセンブラは，JMP 命令に対してディスプレースメント付きプログラム・カウンタ・リラティブモードを使用します．

もう1つのプログラム・カウンタ・リラティブモードは，これと類似していますが，アドレスを計算するとき，レジスタの内容も加算します．このようなレジスタを，インデックス・レジスタといい，16個のレジスタのうちどれでも使用することができます．インデックス・レジスタのどこまでが有効であるかを示すために，レジスタ名にサイズ指示子〝.W〟(デフォルト）または〝.L〟を付けます．このモードでのディスプレースメントは8ビット長のみですが，符号付きなので，PC 値を〔−128〕〜〔+127〕の範囲で修飾することができます．

このモードを使用する最も一般的な場合としては，実際にジャンプする位置をプログラムの前の部分であらかじめ決定しておき，いくつかの位置の１つへジャンプする場合があります．リロケータブル記号のあとに，カッコで囲んだ(データまたはアドレス)レジスタ名を付けた場合，このモードが選択されます．例を次に示します．

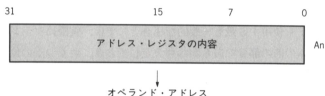
```
          RORG    0
*          :                    Code which calculates
*          :                    which routine should be
*          :                    executed, and places
*          :                    0, 4, 8, or 12 in A0
*          :                    accordingly. (We know
*          :                    each JMP is 4 bytes long.)
*
```

```
        JMP     JTABLE(A0)     Jump to appropriate JMP
*                              instruction
*
JTABLE  JMP     ROUTINEA       Executed if A0 contains  0
        JMP     ROUTINEB       Executed if A0 contains  4
        JMP     ROUTINEC       Executed if A0 contains  8
        JMP     ROUTINED       Executed if A0 contains 12
```

このモードには，インデックス・レジスタの16または32ビットのいずれかを使用するかによって2つの異なる形式があります．

```
        JMP     JTABLE(A0.W)  is the same as in the
*                             example above. Only the
*                             bottom 16 bits of A0 used
*                             treated as a 16-bit signed
*                             number.
        JMP     JTABLE(A0.L)  uses all 32 bits of A0.
```

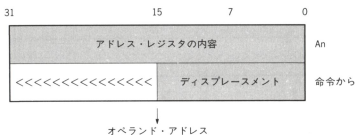
○アドレス・レジスタ間接

アドレス・レジスタ上のオペランド・アドレスに基づいた，5種類のアドレッシングモードがあります．レジスタ自体がオペランドではなく，メモリ内のオペランドを指しているので，このモードのことを**間接**アドレッシングと呼びます．

これらのモードの最も単純な形式は，次のようにアドレス・レジスタ名をカッコで囲んで書くことによって指定します．

```
CLR.B    (A2)
```

これはA2が指す番地の1バイトをクリアします．

○ディスプレースメント付きレジスタ間接

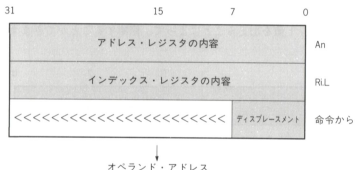
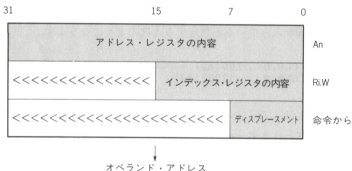
アドレス・レジスタに入っているアドレスは，符号付きの16ビット・ディスプレースメント値を加えることによって，修飾することができます．

A2 に50000が入っていると想定すると次のようになります．

```
CLR.B   100(A2)       Clears byte at 50100
CLR.B   -32000(A2)    Clears byte at 18000
```

このモードは，アドレス・レジスタが，あるデータ構造へのポインタである場合などで使われます．そして，そのデータ構造では，各項目はポインタからの固定的なオフセットとして参照されます[注2]．

○ディスプレースメントおよびインデックス付きレジスタ間接

これは，前に説明したモードと類似していますが，他のレジスタの内容を加算することと，ディスプレースメント値のサイズが8ビットのみである点が違っています．

ロング形式

ワード形式

インデックス・レジスタは，16個のレジスタのうちどれでも使用することができます．インデックス・レジスタのどこまでが有効であるかを示すために，レジスタ名にサイズ指示子〝.W″（デフォルト）または〝.L″をつけます．A0に＄230000，A1に＄FFFC(16ビット数の－4に等しい)，そしてA2に＄20が入っていると想定すると次のようになります．

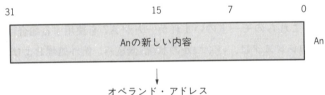
```
CLR.B    $10(A0,A2)      clears byte $230030
CLR.B    $10(A0,A2.W)      "      "  $230030
CLR.B    $10(A0,A2.L)      "      "  $230030
CLR.B    $10(A2,A0.L)      "      "  $230030
CLR.B    $10(A2,A0.W)      "      "      $30
CLR.B    $10(A0,A1.W)      "      "  $23000C
CLR.B    $10(A0,A1.L)      "      "  $24000C
```

このモードを使用することにより，レジスタが指しているデータ構造中の，計算されたオフセット位置へアクセスすることができます．

○プレデクリメントまたはポストインクリメント付きレジスタ間接

スタック(第4章を参照)の管理を容易にするため，基本的アドレス・レジスタ間接モードに対して2種類のバリエーションが準備されています．これらはいずれも，レジスタが指す番地を参照し，レジスタの内容を変更して，隣接する番地を指すようにします．

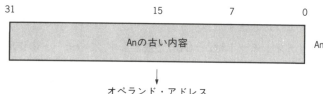
プレデクリメントモードは，－(An)と書きます．このモードの作用は，命令のオペランド・サイズがバイト，ワードまたはロングワードのいずれであるかにしたがって，Anの中の値を1，2または4だけ減算し，調整されたAnによって指される位置へアクセスします．

プレデクリメントモード

したがって，A1, A2, A3の値がそれぞれ1000であるとすると次のようになります．

```
        CLR.B    -(A1)          sets A1 to 999,
*                               and clears the byte at 999
        CLR.W    -(A2)          sets A2 to 998,
*                               and clears the word at 998
        CLR.L    -(A3)          sets A3 to 996,
*                               and clears the long word
*                               at 996
```

ポストインクリメントモードは，(An)＋と書き，プレデクリメントモードと正反対です．Anによってもともと指されている位置にアクセスが行われ，そのあと，Anの値が1，2または4だけ加算されます．

ポストインクリメントモード

A1の値が1000であるとすると，

```
*          CLR.W    (A1)+          clears the word at 1000,
                                   and then sets A1 to 1002
```

となります．これらのモードのいずれか一方がA7を使用する場合は，注意が必要です．このレジスタは，いくつかの状況(割込み，例外処理およびサブルーチン・コール)で，ハードウェアが自動的に使用する，特殊なレジスタで，常に偶数アドレスが保持されているものと見なされます．したがって，これら2種類のモードは，バイト・サイズ命令でレジスタの値を偶数に保つために，A7の値を1でなく，2ずつ調整します．

○イミディエイト・データ

このアドレッシングモードでは，命令それ自体にオペランド値を持ち，ソース・オペランドについてのみ使用が認められます．データ値は，〝#値〟と書き，このデータが記憶される長さは，命令のデータ・サイズによってきまります．したがって，

```
MOVE.B   #$FF,D0
```

これは16進数 FF を D0 の下位バイトにロードし，

```
MOVE.L   #$56789ABC,D0
```

これは D0 全体を56789ABC(16進数)にセットします．

よくあるプログラミングエラーで，必ずしもすぐに検出されるとは限らないものに，イミディエイト・オペランドの〝#〟を抜かしてしまうケースがあります．もし誤って次のように書いた場合，

```
MOVE.B   $FE,D0
```

結果は D0 に＄FE という値を入れるのではなく，＄FE 番地の内容をロードすることになります．

いくつかの命令には，命令の中に小さいイミディエイト・オペランドを埋め込むことのできる，いわゆる〝速い〟バリエーションがあります．シンタックスは，通常のイミディエイトモードと同じです，例として，8ビットの符号付きオペランドを取る，MOVEQ 命令を示します．

```
MOVEQ    #-3,D7         Set D7 to -3 (size is Long)
```

類似の命令に，1～8の範囲の数を加算または減算する命令があります．例を次に示します．

```
ADDQ.L   #4,A2
SUBQ.B   #1,(A1)
```

○アドレッシングモードのまとめ

これまでに説明したアドレッシングモードを簡単にまとめます.

モード	シンタックス	実効アドレス
データ・レジスタ	Dn	EA＝Dn
アドレス・レジスタ	An	EA＝An
アブソリュート・アドレス	数またはASYMB	EA＝固定的な数 （16または32ビット）
PCリラティブ	RSYMB	EA＝〔PC〕＋d16
インデックス付きPCリラティブ	RSYMB(Ri)	EA＝〔PC〕＋〔Ri〕＋d8
レジスタ間接	(An)	EA＝〔An〕
オフセット付きレジスタ間接	d16(An)	EA＝〔An〕＋d16
インデックス及びオフセット付きレジスタ間接	d8(An, Ri)	EA＝〔An〕＋〔Ri〕＋d8
プレデクリメント・レジスタ間接	−(An)	An：＝An−N；EA＝〔An〕
ポストインクリメント・レジスタ間接	(An)＋	EA＝〔An〕；An：＝An＋N
イミディエイト・データ	＃数または＃ASYMB	命令内のオペランド

記号の説明：

EA＝実効アドレス
Dn＝データ・レジスタ
d8＝8ビットのディスプレースメント
PC＝プログラム・カウンタ
〔　〕＝"～の内容"
ASYMB＝アブソリュート・シンボル
Ri＝任意のＡまたはＤレジスタ
An＝アドレス・レジスタ
d16＝16ビットのディスプレースメント
N＝1，2，4のいずれか（サイズによる）
：＝"代入"
RSYMB＝リロケータブル・シンボル

○インプリシット・アドレッシング

これは前述のまとめのような一般的なアドレッシングモードではありませんが，オペランドを指定するもう１つの方法です．特定のレジスタまたはスタック位置を自動的に使用するいくつかの命令では，オペランドに対するインプリシット参照が発生します．インプリシットに使用することができるレジスタは，プログラム・カウンタ（PC），プロセッサ・ステータス・レジスタ（SR），そして，

アドレス・レジスタ7の2種類の形式(**USP** および **SSP**)であるスタック・ポインタ・レジスタ(**SP**)です.

すでに説明した例の中で，インプリシット・アドレッシングを行うものに，**JMP** 命令があります．これは，ジャンプを反映させるため，プログラム・カウンタを変化させます．

2.6 実効アドレスの分類

オペランドを実効アドレスとして指定する命令の多くは，前述のアドレッシングモードをすべて使えるとは限りません．禁止されているモードを使うと，まったく無意味な場合もあり，単に望ましくないだけの場合もあります．以下の(違法な)命令について考えてみましょう．

```
JMP     D6          Jump to a register
JMP     -(A5)       Decrement A5 by 1, 2, or 4?
MOVE    D4,#77      Copy D4 into constant 77
```

任意のオペランドに対する制約事項を簡潔に表現するために，各種のアドレスモードを，**データ**参照，**メモリ**参照，**可変**オペランド，および**制御**参照という，4つの重複するカテゴリに分類します．そのため，〝制御アドレッシングモード〟，〝データ可変アドレッシングモード〟などの用語を使います．

データ・オペランドは，アドレス・レジスタの内容を除く，すべてのものを含みます．これに対し，メモリ・オペランドは，どのレジスタにも収容されていないものです．オペランドは，書込みが可能であれば可変です．制御オペランドは，ジャンプのデスティネーションを示すために使用することができるオペランドです．

以下の表に各モードが属するカテゴリをまとめます.

モード	データ	メモリ	制御	可変
Dn	*			*
An				*
d16(PC)	*	*	*	
d8(PC, Ri)	*	*	*	
(An)	*	*	*	*
d16(An)	*	*	*	*
d8(An, Ri)	*	*	*	*
−(An)	*	*		*
(An)+	*	*		*
アブソリュート	*	*	*	*
#データ	*	*		

監訳者注

注１： RORGに関する補足

本書で使用するモトローラ標準のアセンブラでは，ORGまたはRORGディレクティブが省略された場合には，〝RORG　0″が指定されたものとみなしている．

例えば，次の例で〝JTABLE″をラベルとするとき

```
ORG     0
JMP     JTABLE(A0.W)
JMP     JTABLE(A0.L)
```

は誤りである．なぜならアドレスレジスタ間接モードでは，サイズ指定は許されないからである．

しかし，次のようにORGをRORGに変更すると

```
RORG    0
JMP     JTABLE(A0.W)
JMP     JTABLE(A0.L)
```

は正しくアセンブルされる．これは，RORG指定を行うことにより，次のように

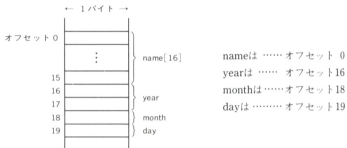
```
JMP     JTABLE(PC,A0.W)
JMP     JTABLE(PC,A0.L)
```

指定したものとみなすからである．

このことをふまえて，3章以降で示される各種のプログラム例で，ORGディレクティブが明示的に示されない場合には，〝RORG　0″が指定されているものとして読み進んでほしい．

注２： 例えばC言語で，

```
struct birthday {
        char name[16];
        int  year;
        char month;
        char day;
};
```

という構造体が，

として表される場合，A0がポインタで1961年1月9日とするには，

```
MOVE.W  #1961,year(A0)
MOVE.B  #1,month(A0)
MOVE.B  #9,day(A0)
```

とすればよい．

データの移動と比較

3.1 単純なデータ移動 66
3.2 条件付き分岐(1) 69
3.3 比較 72
3.4 条件付き分岐(2) 75
3.5 簡単なメモリ・チェック例 77
3.6 ゼロとの比較と移動 79
3.7 小さい数の移動 80
3.8 ビットテスト 81
3.9 条件テスト 82
3.10 ループ制御 83
3.11 簡単な入出力 86
3.12 周辺装置へのデータ移動 89

はじめに

68000命令セットの中で最も基本的な命令は MOVE です．MOVE の目的は，コンピュータ・システムのある部分から他の部分へ，情報を単純に移動することです．他の多くのコンピュータと違って，68000では，レジスタ内部または外部へのデータの移動に区別がありません．もちろん，1つのメモリ位置から他へ，中間的なレジスタを使用せず，直接データを移動することができます．

3.1　単純なデータ移動

基本的 MOVE 命令の変形として多くの命令がありますが，これらについては後述します．まず，メモリをデータで満たすための単純なプログラムについて考えてみましょう．ソースについては，イミディエイト・アドレスモードを使用し，デスティネーションについては，アブソリュート形式を使用することができます．

```
MOVE.B  #123,BYTELOC
```

これは，10進数123をラベル BYTELOC により定義されたメモリ（バイト）にストアします．デスティネーションとしてかわりにレジスタを使い，次の形式を使用することができます．

```
MOVE.L  #123,D1
```

これは，値123をデータ・レジスタ D1 にロードします．この場合，命令のバイト形式ではなく，ロング形式を使用している点に注意してください．データ・レジスタは32ビット幅なので，レジスタ全体を値123にセットしています．例えば，次のような MOVE 命令の別の形式を使用した場合，

```
MOVE.B  #123,D1
```

単に，下位バイトが値123にセットされることになります．この場合，レジスタD1の残りの部分は変化しません．この作用が便利である場合は多くありますが，ただし，最初にレジスタが空であることを確認せずに，誤ってレジスタにバイトを移動してしまうミスも起こりがちです．特に，メモリからデータバイトをレジスタに移動する場合がそうです．次の命令は，

```
MOVE.B  BYTELOC,D1
```

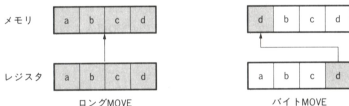
BYTELOCで示される位置に記憶されているバイト値を，レジスタD1全体にはセットしません．そのかわりにBYTELOCのバイト値を，D1の下位8ビットにロードします．D1に記憶されたこの値に対して，後の段階で作用する命令がバイト長命令だけなら，もちろん何の問題もありません．しかし，次のようなケースでこの命令を使用する場合は，注意が必要です．

```
MOVE.B  BYTELOC,D1
MOVE.L  D1,LONGLOC
```

これはLONGLOCにある32ビット値をD1の前の値の上位24ビットにセットし，LONGLOC+3にあるバイトをBYTELOCに記憶されている値にセットします．

もう1つの陥りやすい問題点は，バイト長命令がレジスタの下位8ビットを変化させることを忘れ，メモリ位置の上位8ビットだと思い込んでしまうことです．記憶域への参照は指定されたアドレスから，それを使い始め，LONGLOCへバイトを移動することにより，その位置の1バイトが変化します．

ロングMOVE

バイトMOVE

命令のワード形式を使用した場合,位置 LONGLOC,および LONGLOC+1 の,2バイトが書き込まれます.しかし,命令のワード形式を使って,ワードをレジスタに入れ,次にロング形式を使って,そのレジスタを LONGLOC に記憶すると,LONGLOC~LONGLOC+3の4バイトが変化します.上位2バイトに,レジスタの以前の値が含まれ,下位2バイトに,移動してたきたワード値が入ります.

MOVE 操作のデスティネーションとしては,任意のデータ可変アドレスモードを使用することができ,また,ソースとしては,1つの例外を除いて,任意のアドレスモードを使用することができます.この例外としては,命令のバイト形式を使用する場合であり,このときはアドレス・レジスタをソースとして使用することはできません.

MOVE 命令を使ってデータをメモリまたはデータ・レジスタに移動した場合,ステータス・レジスタ内のコンディション・コードが,それに応じてセットされます.移動されたデータ値が0である場合,ステータス・レジスタ内の Z フラグがセットされ,それ以外の場合はクリアされます.値が負である場合,N フラグがセットされ,それ以外の場合はクリアされます.オーバーフローを示す V フラグ,そして,C フラグ(桁上がりが発生した場合に通常セットされる)は,両方ともクリアされます.X フラグは,直前の算術命令で桁上がりが発生したことを覚えておくために使用され,したがって,このステータスフラグは変化しません.

アドレス・レジスタに値が移動される場合には,ステータス・レジスタはこのような変化は起きません.その理由は,あとの命令でテストされる可能性のあるコンディション・コードを変化させずに,インデックスとして使用されるアドレス・レジスタの値を調整するのが便利だからです.この違いを示すために MOVEA(MOVE Address,アドレスの移動)という独立した命令を使って,データをアドレス・レジスタに移動します.実際には,命令のオペコードは,MOVE の場合と同じです.多くのアセンブラでは MOVEA を指定せず,アドレス・レジスタに対して単に MOVE を使用することが認められています.ただし,コンディション・コードがセットされないことをユーザー自身が覚えておく手段として,必要な場合には MOVEA を使用することは,賢明だと言えます.

多くの連続する領域を同じ値にセットしたい場合，次のプログラムによってこの処理を行うことができます．

```
ORG      $1000
MOVEA.L  #$2000,A0     Load start address
MOVE.L   #0,D2         Load value
MOVE.L   D2,(A0)+      Store value and move ...
MOVE.L   D2,(A0)+      .. to next location
MOVE.L   D2,(A0)+      and again
END
```

プログラムの1行目は，アセンブルをアブソリュートモードで行い，開始番地を＄1000とします．2行目は，アドレス・レジスタA0に，それ以降ポインタとして使用される値をロードします．このポインタの初期値は＄2000です．同様に3行目は，データ・レジスタD2の全32ビットを，値0にセットします．4～6行目は，レジスタD2の内容を，アドレス・レジスタA0の内容で示される位置に入れます．処理のサイズはロングなので，＄2000～＄2003の4バイトは，ゼロにセットされます．(A0)の後には＋符号が付いているので，アドレス・レジスタは，処理後インクリメント(増分)されます．MOVE命令のサイズがロングなので，レジスタのA0は，4だけインクリメントされます．MOVE.Wと指定していれば，A0は2だけインクリメントされることになります．またMOVE.Bと指定していれば，1だけインクリメントされることになります．

したがって，レジスタA0には＄2004が入っており，5行目はバイト＄2004～2007をゼロにセットし，そしてA0を再びインクリメントして＄2008にします．同様に，6行目はバイト＄2008～200Bをセットし，A0は＄200Cで終了します．

3.2 条件付き分岐〔1〕

MOVE命令が，データを1つの場所から他の場所へ移動するとともに，コンディション・コードをセットすることは，すでに説明したとおりです．このことを利用して，メモリの大きなセクションをクリアするための，小型のプログラムを書くことができます．それほど面白いプログラムではありませんが，たった2種類の命令だけで処理を行うことができます．ここで必要となる新しい

命令は，ほとんどのコンピュータで見られる条件付き分岐です．

条件付き分岐は，1個ないし複数のコンディション・コードをテストし，そのコンディション・コードがセットされているか否かに応じて，プログラムの他の部分にジャンプする命令です．各種のコンディション・コードに対応して，いくつかの条件付き分岐命令があります．まず，BEQおよびBNEについて説明します．前者は，Zフラグがセットされているときに指定の位置へジャンプし，そうでないときは，BEQの次の命令が実行されます．BEQは，〝ゼロに等しい場合に分岐〟の意味です．同様に，後者は〝ゼロに等しくない場合に分岐〟の意味でZフラグがセットされていないときに，分岐を行わせます．

これらの限られた命令を使って，指定の番地からゼロ番地までのメモリをクリアするプログラムを作成することができます．

```
        ORG      $1000
        MOVEA.L  #$100,A0     Set up initial pointer
LOOP    MOVE.L   #0,-(A0)     Step pointer down and zero
        MOVE.L   A0,D1        Move pointer into D1
        BNE      LOOP         and loop back until
*                             pointer is zero
        END
```

前例と同様に，1行目はプログラムの開始番地を設定し，次の行でポインタ・レジスタを初期設定します．3行目は，イミディエイト・データ0を，アドレス・レジスタA0によって示される番地にストアします．MOVE命令のサイズはロングであり，アドレス・レジスタはプレデクリメントモードで使用されているので，A0の中の値は，命令が実行される前に，4だけデクリメント(減分)されます．命令が最初に実行されるときは，A0には$FCが入り，$FC～$FFの領域は，ゼロにセットされます．$100番地のバイトは，変化しない点に注意してください．

4行目は，単にポインタの値をデータ・レジスタD1に移動しているだけなので，一見奇妙に思えるかもしれません．しかし，デスティネーションがアドレス・レジスタである場合を例外として，すべてのMOVE命令はコンディション・コードをセットする，ということを思い出してください．したがって，A0が値0になった場合，この処理の後Zフラグがセットされます．1回目はA0は$FCになるので，Zフラグはセットされません．すなわち5行目で，制御が3行目(LOOP

というラベル付き)に戻ります．そして再び，A0 がデクリメントされて＄F8 になり，$F8～$FB の領域は 0 にセットされます．A0 はまだ 0 ではないので，4 行目では Z フラグはセットされず，再びループが行われます．ループは A0 が＄4 になるまで続けられます．A0 が＄4 になり，デクリメントされると 0 になります．領域 0～3 は，3 行目によってクリアされますが，4 行目は 0 という値を A0 から D1 へ移動します．このため，Z フラグがセットされ，分岐は行われずプログラムは終了します．

この小型のプログラムは，いろいろな方法で改良することができます．1 つの方法は，3 行目を次のように変えることです．

```
MOVE.B  #0,-(A0)
```

これによって A0 により示される 1 バイトが 0 にセットされます．この場合，命令はバイト長であり，A0 は，処理前に 1 だけデクリメントされることになります．プログラムの作用は，前とまったく同じですが，ループを回るたびにセットされるのは，4 バイトではなく，1 バイトだけです．このため，ループの回数が 4 倍になるので，実行時間が長くなります．バイト長の MOVE 命令が，ロング長の命令よりも実行時間が短いので，全体の実行時間はその 4 倍より若干短くなります．

本当の違いはプログラムがより短くなるということです．というのは，イミディエイト・データが，プログラム内で 2 ワードではなく 1 ワードで収容されるようになるからです．この場合，余分の 2 バイトは速度が増加するということに対する安価な代償です．ただし，ほとんどの演算につきものの問題として，常に記憶空間の大きさをとるか速度をとるか，という選択を迫られます．

速度を犠牲にせずにプログラムを小型化する 1 つの方法は，次のように BNE 命令を変更することです．

```
BNE.S    LOOP
```

条件付き分岐命令にはすべて長い形式と短い形式があります．どちらの形式でも，命令中に記憶される値はジャンプすべき実際の位置ではなく，プログラム内の現在位置から，要求されるラベルへの距離を表す符号付き数です．長い方の形式はこのディスプレースメントを記憶するために2バイトを使用し，短い方の形式は，1バイトしか使用しません．ニーモニックの後に〝.S″を置くことによって，アセンブラに対し，分岐命令の短い方の形式を使用するように指示します．この形が使用できるのは，分岐するべきラベルと分岐命令との距離が，128バイトより小さい場合だけです[注1]．〝.S″で修飾しない形式では最高32767バイト前または後への分岐を行うことができます[注2]．ある種のアセンブラは，後方への分岐については自動的に短い方の形式を使用しますが，未宣言のラベルに対する分岐の場合，短い方の形式を特に指定しない限り，アセンブラでは常に長い方の形式を使用します．

3.3 比 較

前節では，MOVE命令によってコンディション・コードがセットされる，という事実を利用しました．この事実は一時的に値をどこかへ移動したい場合に便利ですが，値が0または負であることをチェックする場合にも便利です．2つの値を比較したい場合が多くありますが，この処理を行うのがCMP命令です．

CMPの一般的な用途は，2つの値が同じであることを調べることです．CMP命令で使用した2個のオペランドが等しい場合，Zフラグがセットされます．したがって，次のプログラムは，

```
CMP.L   D0,D1
BEQ     EQUAL
```

D0とD1に同じ値が入っている場合にラベルEQUALへのジャンプを行います．CMPの実際の動作は，第2オペランドから第1オペランドを減算し，それに合わせてコンディション・コードをセットすることです．この減算の実際の結果は捨てられ，第2オペランドのもとの値は変化しません．コンディション・コードは，変化しないXフラグを例外として，すべて，セットされるかクリアさ

れるかのどちらかです.

CMP 命令には，4つの形式がありますが，ほとんどのアセンブラでは，自動的に適切な形式を選択します．CMP 形式はデスティネーション・オペランドとしてのデータ・レジスタについてのみ，使用されます．比較される値は，バイト，ワード，またはロングを使って指定することができます．ソースとしては任意のアドレスモードを使用することができますが，1つの例外があります．その例外とは，サイズをバイト指定する場合で，この場合，ソースをアドレス・レジスタの中に置くことはできません(間接アドレッシングを使って，すなわちポインタとしてバイト値を指す場合は問題ありません)．次に，有効な例を示します．

```
CMP.B   12(A3),D0
```

これは A3 によって示される位置から，オフセット12のところにあるバイトと，D0 の下位 8 ビットを比較します．

命令の CMPA 形式は，デスティネーション・オペランドとしてアドレス・レジスタについてのみ，使用することができます．この場合，値はワードまたはロングのいずれかでのみ，指定することができ，ソースとしては任意のアドレスモードを使用することができます．命令のワード形式を使用した場合，指定された値は，32ビットまで符号拡張され，その結果のロング値を比較に使用します．したがって，

```
CMPA.W  #$FFFF,A2
```

これは A2 が−1($FFFFFFFF)に等しい場合に，Z フラグをセットし，A2 に$0000FFFF が入っている場合にはセットしません．

CMPI 形式は，データ可変のデスティネーションについてのみ，使用することができます．したがって，アドレス・レジスタの内容や，プログラム・カウンタ相対の値は使用できません．ソースは常にイミディエイト・データでなければなりません．命令は 3 種類の長さのいずれも使用することができます．

```
CMPI.B  #$0A,-(A0)
```

これはA0(に記憶されている値)を1だけデクリメントし，値$0Aを，A0の新しい値によって示されるバイトと比較します．CMPIは，デスティネーション・オペランドとしてのデータ・レジスタについても使用することができます．この場合，ソース・オペランドとしてイミディエイト・データを指定してCMPを使用した場合と同じ作用になります．

CMPの最後の形式は，メモリ位置同士の比較に使うもので，CMPMと指定します．この場合，ソースおよびデスティネーション・オペランドは，ポストインクリメント・アドレスモードによってのみ，指定することができます．比較はバイト，ワード，またロングワードについて行うことができます．CMPMは，メモリの大きな区分を比較する場合に便利です．以下のプログラムは，$1000番地から始まる100バイトのメモリと，$2000から始まる100バイトを比較するものです．

```
        MOVEA.L  #$1000,A0     Load first pointer
        MOVEA.L  #$2000,A1     Load second pointer
LOOP    CMPM.B   (A0)+,(A1)+   Do comparison
        BNE.S    NOTSAME       Jump if not equal
        CMPA.L   #$1064,A0     Check end condition
        BNE.S    LOOP          Loop back if more to do
```

まずここでは，比較したいメモリ領域に対するポインタとして，2つのアドレス・レジスタをロードします．3行目は，アドレス・レジスタによって示される2つのバイトを比較し，ポインタをインクリメントします．2つの値が等しくない場合，4行目でループから抜けだします．2つの値が等しい場合，続けて次の2つのバイト(各バイト)を比較しなければなりません．アドレス・レジスタはすでにインクリメントされて，次の比較の用意ができていますが，まず，すべてのバイトが検証されたかどうかをチェックしなければなりません．5行目は，第1のポインタと，ベース・アドレス+100を比較します．A0がまだこの値に等しくない場合は，6行目でラベルLOOPに戻り，次のバイトの組を調べます．それ以外の場合は，このプログラムを終えて，2個の100バイト領域が同じであることがわかります．

3.4 条件付き分岐〔2〕

これまでは，2種類の条件をテストする条件付き分岐についてのみ説明しました．それは，Zフラグがセットされているときに分岐するBEQと，Zフラグがセットされていないときに分岐するBNEです．想像できるとおり，Bcc命令には，この他にも多くの形式があり，他の条件をテストします．

これらの命令の1番目のグループは，ステータス・レジスタ中の1ビットによってのみ制御されます．BEQとBNEが，Zフラグの値にしたがって分岐を生じさせるのと同様に，BCSとBCCは，キャリーフラグ(C)の状態をテストするのに使用することができます．BCSは〝キャリーフラグがセットされているときに分岐〟の意味で，Cフラグが現在セットされているときに分岐します．BCCは〝キャリーフラグがクリアされているときに分岐〟の意味で，Cフラグがセットされていない場合に分岐します．

BMIとBPLは，まったく同じ方法でNフラグをテストするために使用することができます．BMIすなわち〝マイナスのとき分岐〟は，Nフラグがセットされている場合に分岐が行われることを意味します．BPLすなわち，〝プラスのとき分岐〟は，Nフラグがセットされていない場合にだけ分岐が行われることを意味します．値が0の場合にNフラグはクリアされるので，BPLはこの場合でも分岐を行います．

1番目のグループの最後の組はBVSとBVCで，オーバーフローフラグ(V)がセットまたはクリアされているときに分岐を行います．

条件付き分岐の2番目のグループでは，分岐を行うかどうかを決定する前に，多くの条件をテストします．これらの命令の中のいくつかは，前述のより単純なテストについては非常によく似ており，唯一の違いは，オーバーフローおよびキャリーフラグの取扱いだけです．MOVEなどの多くの命令は，常にCとVをクリアするため，このような場合には，2つの形式は同一です．両者の違いが重要となるのは，符号付きの数を処理する場合です．

BLTとBGEは符号付きの数を比較する場合に使用し，それぞれ〝より小さいときに分岐〟および〝より大きいか等しいときに分岐〟の意味です．BLTは，

BMI と同じ方法で N フラグをテストしますが，ただし N フラグがセットされていてオーバーフローフラグ V がセットされていない場合にのみ，分岐を行います．V がセットされている場合は，N フラグもセットされていない場合に，分岐を行います．すなわち，オーバーフローが発生しない限り，BLT は BMI と同じ動作をします，オーバーフローが発生すると，BLT は BPL と同じ動作をします．BGE も N および V フラグをテストし，両方ともセットされていないか，あるいは両方ともセットされている場合に，分岐を行います．この点で，BGE は，オーバーフローが発生しない場合は BPL と，オーバーフローが発生した場合は BMI と同じ動作をすると言えます．

BLS と BHI は，Z および C フラグをテストします．BLS は〝より小さいか等しいときに分岐〟の意味で，C フラグまたは Z フラグのいずれかがセットされている場合に，分岐をおこないます．BHI は〝より大きいときに分岐〟の意味で，C および Z が両方ともセットされていない場合にのみ，分岐を行います．BCC と BCS は場合によっては，BHS と BLO とも呼ばれ，それぞれ〝より大きいか等しいときに分岐〟および〝より小さいときに分岐〟の意味です．

最も複雑な分岐は，〝より小さいか等しいときに分岐〟(BLE)と，〝より大きいときに分岐〟(BGT)です．BLE は，BLT でテストされる条件が真である場合に分岐を行いますが，さらに，Z フラグがセットされている場合にも分岐を行います．BGT は，BGE と同じテストを行いますが，分岐が発生するためには，オーバーフローが発生したか否かに関わらず，Z フラグはセットされていてはなりません．

DBcc 命令と Scc 命令で，ステータスビットの同じ組合わせをテストするために，同じ条件名を使用する方法については，あとで説明します．これらの命令とともに，追加的な条件 T(True, 真)および F(False, 偽)を使用することができます．BT(真のときに分岐)と同じ意味で，もちろん BRA と書くことができます．BF(分岐を行わない)と同じ意味の命令はなく，この(可能な)組合せは，替わりに BSR によって使われます[注3]．

3.5 簡単なメモリ·チェック例

これまでに説明した命令を使って，簡単なメモリ・チェック・プログラムを書くことができます．

まず，メモリ内の1つの領域を取り，ここに一定のビット・パターンを書き込みます．次に，メモリが書き込まれた値を保持しているかどうかをチェックします．これは，基板上のRAMチップがすべて正しく作動しているかどうかを調べる場合に便利です．

入出力処理の方法についてはまだ説明していないので，プログラムが何らかのエラーを発見した場合は，単に一定の番地へジャンプします．これはユーザーに対してメッセージを書き出すモニタ・ルーチンの番地と考えられますが，詳細はここでは必要ありません．

```
        ORG     $400
* Define some useful constants
MEMLO   EQU     $1000           lower limit
MEMHI   EQU     $2000           upper limit
TPAT    EQU     $AA             test pattern
MONLOC  EQU     $2000           monitor return address
*
* The memory check program
*
ENTER   MOVEA.L #MEMLO,A0       Set up base pointer
* Fill memory with required value
LP1     MOVE.B  #TPAT,(A0)+     Store value, increment A0
        CMPA.L  #MEMHI,A0       Check limit reached
        BNE.S   LP1             No, keep going
* Check memory has kept that value
        MOVEA.L #MEMLO,A0       Reset base pointer
LP2     CMPI.B  #TPAT,(A0)+     Check value is the same
        BNE     MONLOC          Not the same - error at
*                               (A0)-1
        CMPA.L  #MEMHI,A0       Check limit reached
        BNE.S   LP2             No, keep going
* Check complete. Go back and try it again
        BRA.S   ENTER
        END
```

最初の数行は，プログラムの開始番地の定義と，EQU ディレクティブを使って，いくつかの値を定義します．EQU を使うと，あとの段階でのプログラムの変更がより簡単に行え，特定の値に対して名前を定義するために EQU を使うことは良い習慣だと言えます．例えば，このプログラムでは，$1000～$1FFF のメモリをテストします．これらの値は，ラベル MEMLO および MEMHI で定義しています．メモリの別の領域をテストするようにプログラムを変更したい場合は，プログラムそれ自体が特定の数の使用個所を探し，それらの値を変更するのではなく，ただ単に EQU 文を変更するだけで簡単に行うことができます．

プログラムはラベル ENTER から始まり，ここで A0 は，テストの対象となるメモリ領域の開始番地にセットされています．ラベル LP1 はループの開始番地を定義し，A0 によって示されるバイトに，TPAT で定義されるテスト・パターンを書き込み，A0 をインクリメントします．次の行は CMPA を使って，必要なメモリをすべて TPAT で満たしたかどうかをチェックします．もし満たしていなければ，LP1 へ戻ります．メモリがすべて満たされている場合にのみ，このループから抜け出ます．

メモリが TPAT で満たされると，ポインタ A0 をリセットし，再びテスト領域全体をループします．メモリに記憶されている値が，期待されているとおりの値ではない場合，MONLOC へ分岐します．レジスタ A0 はすでにインクリメントされているのでエラーのある実際の番地は，A0 に入っているアドレスから 1 を引いたものとなります．

MOVEA.L と CMPA.L を使用している点に注意してください．この例では，MOVEA.W と CMPA.W も同じ動作をし，またその方がプログラムが短くなります．ただしその場合，プログラムをあとで保守，または変更した場合には，厄介な落とし穴が残る可能性があります．チェックするメモリの上限を，$2000 から $8000へ拡張したい場合を考えてみましょう．この変更を行う人は，プログラムを見て MEMHI の定義を $2000を $8000へ変えればいい，と思うことでしょう．MOVEA と CMPA のワード長形式を使用した場合，ループの終わりのテストが行われるとき，プロセッサは MEMHI によって定義されるイミディエイト値を取り，それを32ビットまで符号拡張し，A0 との比較を行います．このため，A0 の値が，$FFFF8000のときにのみループが終了することになりますが，これは有効なアドレスではありません．実際には，有効なメモリがすべて満たされ

るとすぐに，プログラムはバスエラー注4のために停止します．この例の教訓として，アドレス・レジスタにアドレス値を入れる場合は常に，命令のロング形式を使用した方が良い，ということを覚えておいてください．他の形式は一般的に，アドレス・レジスタにデータ値を入れる場合にのみ，使用するようにします．

3.6 ゼロとの比較と移動

68000の命令の中には〝0〞の取扱いに使用する2つの特殊な命令があります．すでに説明したように，MOVEをイミディエイト・ソース・データに対して使用することができます．また任意の値をメモリまたはレジスタに移動でき，この値が0であってもさしつかえありません．同様に一連のCMP命令もイミディエイト・データに対して使用することができ，それが0でもいいわけです．ただし，イミディエイト値は，16ビットの命令ワードの次の，1個ないし2個の拡張16ビット・ワードとして表現され，したがって次の処理では，

```
MOVE.L  #0,D0
```

命令のために16ビットを使い，0というロング値の表記のために32ビットを使います．値を0にセットするのは，非常に一般的なことなので，命令長が16ビットで済む2つの特殊な命令があります．

1つはCLRで，指定されたデスティネーションを0にクリアします．このデスティネーションはデータ可変でなければならず，したがって，アドレス・レジスタを0にクリアするためには使用できません注5．ただし，直接またはアドレス・レジスタによって参照されるメモリ内のバイト，ワードまたはロングワードは，0にセットすることができます．同様に，データ・レジスタの下位8, 16, または32ビットも0にセットすることができます．

MOVEを使ってデスティネーションに0をいれた場合と同様に，コンディション・コードがセットされるため，Xは変化せず，Zはセットされ，他のフラグはクリアされます．

メモリ位置が実際にI/Oページの一部であり，メモリ・マップされた装置がメモリ位置と同様に現れている場合は，この命令は注意して使用しなければなりません．この命令は，書込みを行う前に，実際にメモリを読み込みます．そのため，I/Oポートを読み込む動作によって，関連する周辺装置が影響を受ける場合は，奇妙な作用が出る可能性があります．

同様に，TST命令を使って値が0に等しいかどうかをテストすることができます．この場合も，デスティネーションは任意のデータ可変オペランドを指定することができ，サイズはバイト，ワードまたはロングのいずれでも構いません．指定された値が0に等しい場合はZフラグがセットされ，それ以外の場合はクリアされます．このあとには，適切なBEQ，BNE，またはBLE命令などが続きます．

さらにTSTを使って，値が負であるかどうかを調べることができます．負である場合は，Nフラグがセットされ，そうでない場合はクリアされます．Xフラグは変化せず，VおよびCフラグは常にクリアされています．したがって，BMIおよびBPLを使って，Nフラグの条件をテストすることができます．この命令のあとでは，キャリーフラグは常にクリアされているため，BLTはBMIと同じ作用をします．同様にこの場合にはBGTとBPLも交換可能です．

3.7　小さい数の移動

68000が他のマイクロプロセッサより優れている特徴の1つが，32ビット値の処理機能なので，多くのプログラマーは，可能な限り命令のロング形式を使うことを希望するでしょう．そのような場合には，レジスタを0または小さい数に初期設定する必要がよく生じます．0に設定するには，すでに説明したように，CLR命令を使ってレジスタまたはメモリ位置を0にクリアすることができ，これは3種類のサイズのいずれにも適用できます．

レジスタを小さい整数に初期設定するには，MOVE命令のロング形式を使って，イミディエイト・データをレジスタに移動します．唯一の問題は，この命令が6バイト(MOVE命令のための2バイト，イミディエイト・データのための4バイト)を使用するということです．イミディエイト値が実際にロング形式で

あれば，明らかに4バイトすべてがその値を入れるのに必要です．しかし，イミディエイト値が1バイトで収まる場合に，0であるバイトを単に入れるだけのために，これだけの空間を使うのは無駄なことだと言えます．

このような状況に対処するために，MOVE命令の特殊な形式が準備されています．MOVEQ(Move Quick，素早く移動の意)命令は，サイズがロング形式のみで，1バイトに収まる数をデータ・レジスタに移動する場合にのみ，使用することができます．作用はMOVEを使って，〔−128〕~〔+127〕の範囲のイミディエイト値をデータ・レジスタに移動する場合と同じですが，MOVEQ命令は2バイトのみを必要とする点が異なっており，命令の下位バイトにイミディエイト値がパックされます．必要に応じてデータは符号拡張されるのに伴い，データ・レジスタ全体が変化します．ロードされた値が負または0の場合，NまたはZステータスフラグがセットされます．VおよびCフラグは常にクリアされ，Xは変化しません．

MOVEQ命令は，データ・レジスタに対して作用するCLR命令のロング形式よりも，少ない実行時間で済みます．したがって，データ・レジスタ全体を0にクリアするためのより良い方法であると言えます．MOVEQは常にサイズがロング形式であり，データ・レジスタに対してのみ使用することができます．

3.8 ビットテスト

単一のビットに対して使用できる多くの命令があります．これらについては，第6章で詳細に説明しますが，ここでは1つの命令について説明します．それはBTST命令で，デスティネーション中の特定のビットをテストするために使用します．ビットが0に等しい場合，Zフラグがセットされます．ビットが1である場合，Zフラグはセットされません．これ以外のコンディション・コードはいずれも変化しません．

デスティネーションの位置は何の影響も受けず，任意のデータ・アドレッシングモードを使って指定することができます．デスティネーションがメモリかデータ・レジスタのいずれであるかに応じて，命令の作用が変わります．メモリの場合，メモリから1バイトが読み込まれ，そのバイト内のビットがテスト

されます．最下位ビットをビット0と指定し，最上位ビットを7と指定します．7より大きい数は，8の剰余と見なされます．

データ・レジスタをデスティネーションとして使用した場合，ビット番号は0～31の範囲となり，レジスタ内の全ビットをテストすることができます．31より大きい数は，やはり32の剰余と見なされます．したがって BTST 命令のサイズは，プログラマが指定するのではなく，デスティネーション・オペランドに応じて，バイトとロングの間を変化します．

ソース・オペランドとしてビット番号を指定しますが，それは2つの方法で指定することができます．第1の方法はイミディエイト形式を使用することです．この場合，指定した値はビット番号として使用されます．第2の方法は，データ・レジスタを与えることで，この場合，プロセッサは，データ・レジスタに収容されている数値をビット番号として使用します．どちらの場合でも，デスティネーションがメモリまたはデータ・レジスタのどちらであるかに応じて，ビット番号は8または32の剰余が使用されます．

ここで注意しておくべきことは，テストするべき特定のビットを示すビット・パターンではなく，ビット番号それ自体が使用される，ということです．

3.9 条件テスト

これまでに，一連の Bcc 命令を使って，コンディション・コードのさまざまな組合せの状態に応じて分岐を行う方法について説明してきました．これが最も一般的なコンディション・コードの使用法ですが，この他に，コンディション・コードを検査するために2つの命令があります．1つは Scc(コンディション・コードに応じてセット)です．この命令は，一連の Bcc と同じ条件の集合を使って，1個ないし複数のコンディション・コードの値をテストします．条件を満足すれば，デスティネーションとして定義されたバイトが，$FF にセットされます．条件を満足しない場合，デスティネーション・バイトは0にセットされます[注6]．したがって，例えば，

```
        Scc     BYTELOC     Set BYTELOC according to
*                           condition
```

これは次のプログラムと同じ意味を持った単一の命令です.

```
        Bcc      NXT          Branch if condition
*                             satisfied
        CLR.B    BYTELOC      Clear BYTELOC, not
*                             satisfied
        BRA.S    NXT1         Branch to end
NXT     MOVE.B   #$FF,BYTELOC Set BYTELOC to $FF
NXT1    ....     Rest of program
```

Sccは,単一のバイトをセットする場合にのみ,使用できる点に注意してください.このバイトは,データ可変アドレッシングモードを使って指定しなければなりません.また,バイトを常に$FFにセットする,便利な方法でもあります.というのは,TRUEの条件テストを使用した場合,

```
        ST       BYTELOC
```

BYTELOCの中の全ビットが無条件にセットされるからです.FALSE(SF)を使った同じテストはCLR命令のバイト形式を使用した場合とまったく同じです.

Scc命令は一般的にあとの段階でのテストのために,特定のコンディション・コードの状態を覚えておくのに便利です.

3.10 ループ制御

コンピュータで実行される最も一般的な動作の1つに,一連の命令を何度も繰返し実行する動作があります.このループは通常,繰返し変数によって制御され,一定の値に達するまでこの変数がインクリメント(増分)されます.

68000では,ループ制御を補助するための命令を準備していますが,この命令は,一般的に必要とされるのと逆の方向へ作用します.すなわち,繰返し変数を**デクリメント**(減分)します.また,変数が0になったときではなく,負になったときに反復が停止するので,混乱をひき起こすかもしれません.

この一連の命令をDBccといい,〝デクリメントおよび分岐〟の意味です.命令の全動作は,実際にまずコンディション・コードをテストし,条件が満足さ

れない場合にのみ，デクリメントと分岐の部分へ移動します．まず，条件が絶対に満足されない場合の，この命令の使用法を見ることにしましょう．すなわち，DBF，または，条件が偽の場合のデクリメントおよび分岐です．これが最も一般的に使用されるDBccの形式であり，また，ほとんどのアセンブラでは別表記のDBRAを認めています．

DBRAは，ソース・オペランドとしてデータ・レジスタをデスティネーション・オペランドとしてラベルを取ります．この命令のサイズは常にワードです．レジスタの値が0である場合，次の命令が実行されます．それ以外の場合は，レジスタの中の値が1だけデクリメントされ，デスティネーションとして指定されたラベルへジャンプが行われます．

実際には，デクリメントは常にレジスタで発生するため，ループが完了するときには，使用されるレジスタは0にはならないので，前の説明は完全に正確であるとは言えません．また，レジスタの下位16ビットだけが，カウンタとして使用されます．ラベルはDBRA命令の前後どちらにあってもさしつかえありませんが，一般的には前です．次のプログラムについて考えてみましょう．

```
         MOVE.L   #$2000,A1     Set up pointer
         MOVE.W   #19,D0        Set up counter
LOOP     CLR.B    (A1)+         Clear byte and increment
*                               pointer
         DBRA     D0,LOOP       Loop while D0 >= 0
```

ここではD0の下位16ビットがカウンタとして使用されます．カウンタの初期値は19にセットされており，また，アドレス・レジスタA1は，メモリ番地を指すようにセットされます．ラベルLOOPでは，A1によって参照されるバイトは0にクリアされ，また，CLRのサイズがバイトなので，A1は1だけインクリメントされます．DBRA命令はD0をデクリメントし，結果が負であるかどうかを調べます．負でない場合，制御はLOOPへ戻されます．この動作は，D0が0になるまで続きます．このとき，DBRA命令におけるデクリメント動作で負の値(−1)が検出されます．この場合，LOOPへのジャンプは行われず，D0の下位16ビットが$FFFFに，そしてA1が$2014にセットされた状態でプログラムから抜け出ます．

多くの場合において，ループの実行回数は可変であり，繰返しカウントの初

期値が0であれば，ループはまったく行われません．この場合，DBRA命令をループの終わりに置き，ループの開始位置の直前の命令で適切なレジスタに対して繰返しカウントをセットし，ループの終わりで，DBRA命令の無条件分岐を行わなくてはなりません．ここで，カウントから1を引いた値ではなく，繰返しカウントをレジスタに入れなければならない点に注意してください．繰返しカウントが初期的に0である場合，DBRA命令は分岐を生じません．したがって，ループは完全にバイパスされます[注7]．

DBcc命令のDBRA形式は，一般的に最も便利ですが，完全な形式は非常に強力です．この場合，条件を指定し，その条件が真であれば，DBcc命令は何の影響も与えません．直後の命令に，通常の実行が移ります．条件が偽である場合，データ・レジスタがデクリメントされ，その結果が−1でない場合にのみ，指定の分岐が行われます．

これによって，多数の非常に強力なループ構造が可能になります．例えば，あるプログラムで，一定の値に等しいバイトが見つかるまで1つの場所から他の場所へデータをコピーする必要があるとします．デスティネーション領域は長さが限られているため，デスティネーションがいっぱいになると，コピー動作も停止します．このような状況が発生する場合としては，端末装置から内部バッファへ1行の情報を読み込む場合が考えられます．文字〝リターン〟が見つかるとコピーは停止しますが，指定したサイズよりも長い行が入力された場合にも停止します．次のプログラムを使用すると，

```
CR      EQU       $0D          ASCII carriage return
        MOVEA.L   #$2000,A3    Set up pointer to buffer
        MOVE.W    #79,D0       Allow for 80 characters
RCH     ...       read character into D1
        MOVE.B    D1,(A3)+     Save character
        CMP.B     #CR,D1       Check to see if end of
*                              line
        DBEQ      D0,RCH       Loop unless return or
*                              buffer full
```

最初の2行は，A3をバッファへのポインタとして，D0をバッファ・サイズ(単位：バイト)として初期化します．ループを繰り返すたびに，コンソールから何らかの方法で文字を読み込み，それをポストインクリメント・アドレッシングモードを使ってバッファに格納します．最後に，読み込まれた文字と，キャリ

ッジ・リターンを表すASCIIコードと比較します．読み込まれた文字がリターンである場合，DBEQ命令は何の作用もせずループは停止します．入力行の終わりに達しない場合，バッファ内にまだ余地がある場合にのみ分岐して，他の文字を読み込みます．

3.11 簡単な入出力

これまでの例ではすべて，テスト・プログラムがモニタの制御下で実行されると仮定しており，ユーザーは自分のプログラムを入力して実行を開始することができました．一般的に，このようなモニタは，コンピュータに接続されたターミナルへ情報を書き込んだり，ターミナルからの情報を受け入れるメカニズムを装備しています．

また，ユーザーのプログラムをオペレーティング・システムの制御下で実行することもあるでしょう．この場合にも情報を入出力するための何らかのメカニズムが提供されています．いずれの場合でも外界との通信に使用される方法は，何らかの種類の端末装置を接続できる，シリアル・ラインへの接続である場合が多いでしょう．

このシリアル・ライン接続の最も一般的な方法は，ACIA(非同期通信インターフェース・アダプタ)といった，特殊なチップを使用する方法です．ここでの説明は，6850 ACIAに適応されるものですが，ほとんどの入出力チップでも同様の動作をします．このデバイスについてはあまり詳細に説明しません．それは単に，シリアル・ラインを通じて1バイトの情報を送受信するのに必要とされるすべての動作を取り扱うだけです．

68000のメモリ空間の一部に，ACIAがあります．個々のACIAには，コントロール・ポートとデータ・ポートという2つのポートがあります．8ビット・コンピュータでは，これらのポートは，隣接するメモリ番地にあります．68000では，2つの隣接する16ビットワードの下位バイト[注8]にあります．

ACIAは初期的には，リセットされていなければなりません．リセットは，コントロール・ポートに3という値を書き込むことによって行われます．ACIAのインストラクション・マニュアルによると，ACIAをリセットして動作可能にな

るまでの間，しばらく待つ必要があります．

次にシリアル・ラインの特性(パリティ，割込み可能かどうかなど)を選択しなければなりません．まず初めは，ACIA をポーリングモードで使用します．すなわち，68000がポートをチェックして受信文字を調べていない限り，文字が失われる可能性があります．ACIA には，文字が失われたことをユーザーに知らせる能力がありますが，何が失われたかについては，調べる方法はまったくありません．割込みモードでの ACIA の使い方についてはあとで説明[注9]することにして，ここではポーリング・モードについて説明します．

ACIA に対するセットアップモードとして，$15[注10]という値を使用しますが，この値は単にマジック・ナンバーと考えてください．もし必要ならば，ACIA に関する標準の説明書を読んで，個々の値がどういう意味であるかを調べてください．このマジック・ナンバーはコントロール・ポートに書き込まれます．

ACIA がセットされると，コントロール・ポートの下位 2 ビットを使ってデータ・ポートの状態が示されます．シリアル・ラインを通じて文字が受信された場合は，ビット番号 0 が 1 になります．この段階でユーザーはデータ・ポートから文字を読み込むことができ，これによって，次の文字が受信されるまで，ビット 0 はオフ(0)となります．

ビット番号 1 は，ACIA がラインを通じて文字を送れるかどうかを示します．もしこれが 1 の場合，ユーザーはデータ・ポートに文字を書き込むことができ，これがラインを通じて送信されます．この処理には，しばらく時間がかかり，ACIA がこの処理をしている間に，ビット 1 が 0 にセットされます．バイトが伝送されると再びビット 1 が 1 にセットされ，ユーザーは次の文字を送ることができます．文字の送信と受信は完全に独立しており，このため名前の中に〝非同期〟[注11]という言葉が入っています．

ここまでの説明は，少し複雑そうに見えるかもしれません．しかし，68000の命令セットのいくつかはすでに知っていますので，ACIA の操作は実際には非常に簡単です．ターミナルに〝Hello !〟と書き出してみましょう．

```
* Values required by ACIA
A_RST    EQU      $03               RESET Code
A_INIT   EQU      $15               Magic setup value
A_RDY    EQU      1                 Bit set when ready
A_CTRL   EQU      $840021           Ctrl port memory location
A_DATA   EQU      $840023           Data port memory location
*
         ORG      $1000
* Initialise ACIA
ENTER    MOVE.B   #A_RST,A_CTRL Reset ACIA
         MOVE.W   #1000,D0          Initialise counter
WAIT     DBRA     D0,WAIT           Waste time looping back
         MOVE.B   #A_INIT,A_CTRL Set up ACIA
* Send string down the serial line
         MOVEA.L  #STRING,A0        Pointer to string
NXT      BTST     #A_RDY,A_CTRL Test ok to transmit
         BEQ.S    NXT               Not ready yet, try again
         MOVE.B   (A0)+,A_DATA Write byte into data port
         TST.B    (A0)              See if next byte is zero
         BNE.S    NXT               No, loop back to write it
* Data location for string
STRING   DC.B     'Hello!'          Message
         DC.B     0                 Marker at end of string
```

最初の数行は，ACIA に対するリセットと初期設定コード，それにメモリ・マップ内のコントロールとデータ・ポートの番地など，ユーザーにとって便利ないくつかの名前を定義します．ラベル **ENTER** でプログラムを開始しますが，ここでコントロール・ポートにリセット値を入れます．ここでしばらく時間を浪費しなければならないので，D0 を初期設定して，すぐそれを **DBRA** 命令によってデクリメントします．プロセッサは D0 が負になるか，あるいは **DBRA** を1001回実行するまで，同じ命令の先頭へジャンプします．最後に，初期設定に関連するマジック・ナンバーをコントロール・ポートに書き込みます．この段階で文字列を書きだす準備が整いました．

ラベル **NXT** の直前にある命令はラベル **STRING** のイミディエイト値をレジスタ A0 にロードします．プログラムの終りを見れば，ラベルがあるメモリ番地を参照するものとして定義されていることがわかります．その番地は書き出すべき文字列で初期化されています．したがって，A0 はこの段階で文字列の最初の文字を指しています．また，文字列の直後に 0 が置かれていることも，注意しておくべきでしょう．これはメッセージの終りを示すために使用します．

ラベル **NXT** は，**BTST** 命令を参照します．コントロール・ポートのビット 1

が0である場合，ACIA はまだ次の文字を受け入れられる状態ではありません．この場合，コンディション・コードの Z フラグがセットされ，次の行への条件付き分岐によって，戻って再びビットをチェックすることになります．ACIA のコントロール・ポートのビット1がセットされるまでこのループを続け，セットされた段階でユーザーは文字を送信することができます．この処理は，A0 をポストインクリメントモードで使用することによって行い，文字列から文字を出力ポートに送り，同時にポインタをインクリメントします．

最後に，処理が終了したかどうかを調べなければなりません．TST 命令で再びアドレス・レジスタを使用しますが，その値は変化させません．この段階で A0 が，0であるバイトを示していれば，文字列の書出しは終了したことになります．Z フラグがセットされ，TST の次の BNE 命令は通過します．それ以外の場合は，NXT へループして，次の文字を書き出し，ACIA がレディ状態になるまで待ちます．

一般的に，ACIA を使用する前に ACIA がレディ状態になっているかどうかテストした方が良く，また，使用したあとに，再びレディ状態になるまで待たない方が良い，という点に注意してください．ACIA の内部ロジックは，68000 プロセッサとは無関係に作動するので，ACIA で出力が行われている間は，何らかの作業をしている方が良いでしょう．

文字列が書き出されると，プロセッサは，BNE の次の命令を実行しようとします．ここには，文字列〝Hello !〃が書き込まれており，命令としては何ら意味のないゴミが置かれています．一般的な状況ではモニタへ戻るための命令が最後に入りますが，ここでは詳しいことはあまり重要ではありません．

3.12 周辺装置へのデータ移動

前節では，ACIA に2つのポートがあること，そしてそれらが，隣接する2つのメモリワードの下位バイトとして表されることを説明しました．ACIA の仕様では一般的に，ポートはメモリ内の連続するバイトとして示されていますが，ACIA はもともと8ビット・マイクロプロセッサ用に設計されたものです．ACIA が8ビット・マシンに接続されている場合，2つのポートは，メモリ・マップ

内で互いに隣り合う位置にありますが，68000ではそれらは1つおきの番地にあります．68000に接続されるどの周辺装置でも，元来8ビット・マシン用に設計されたものである場合はこのことが当てはまります．これは，68000によって16ビットのデータ・ラインが作られるためです．

多くの場合においてこのことを認めれば，メモリ内の必要とされるバイトを読み書きするのは，非常に簡単なことです．しかし，ある種の条件下においてはこれは不便であり，しかも速度が遅くなる場合があります．この問題に対処するため，MOVE の特殊な形式が準備されています．MOVEP(Move Peripheral，周辺装置との移動)は，データ・レジスタと，アドレス・レジスタおよびオフセットによって指定される番地を取ります．データ・レジスタがソースである場合，そのデータ・レジスタ内に入っている内容は，アドレス・レジスタとオフセットにより指定される番地から始まる，1つおきのメモリ番地に，一度に(1命令での意)入れられます．MOVEP は，ワードまたはロング形式でのみ，使用可能です．次の例をみてみましょう．

```
MOVE.L   #$01020304,D1 Load data
MOVEA.L  #$C00000,A1   Load address register
MOVEP.L  D1,1(A1)      Move data
```

ここで，D1に＄01020304という値をロードし，A1をアドレス＄C00000にセットします．番地＄C00001，＄C00003，＄C00005および＄C00007には，4つの周辺装置コントロール・ポートがマップされていると仮定します．MOVE 命令は上位バイトを取り，これを指定された番地，すなわち＄C00001に置きます．

次に MOVEP 命令は．レジスタから次のバイト(この例では＄02)を取り，次のメモリ番地プラス2，すなわち＄C00003に置きます．次のバイトが次の奇数番地に入り，最下位バイトは番地＄C00007に入ります．

データ・レジスタがデスティネーションである場合は，動作はこの逆となり，メモリの1個おきのバイトが，レジスタに置かれていきます．MOVEP は，いくつかの点で MOVE と非常に違っているので，注意して使用するようにしなければなりません．第1に，MOVE をデータ・レジスタに対して使用した場合，コンディション・コードが変化するのに対し，MOVEP では変化しません．第2に，メモリの1個おきのバイトが転送されますが，これらのバイトは，開始アドレ

スが奇数か偶数かに応じて，奇数バイトまたは偶数バイトのどちらかになります．バイトをアクセスする方法には，特殊なことは何もなく，多くの場合において，バイト・サイズのMOVEがより簡単です．しかし，大量の情報を転送する場合，この命令は便利です．1つの状況として，8ビット・マシン用の浮動小数点プロセッサを68000に取り付けた場合が考えられます．32ビット値すべてが，1個の単純なMOVEP命令によって他のプロセッサへ転送することができます．これに対し，別の方法では，4個のバイト・サイズのMOVEと3個のシフト命令を使わなくてはなりません．

監訳者注

注1：　正確には，前に129，後に126バイトまで可能．

注2：　同様に，前に32769，後に32766バイトまで可能．

注3：　“CMP　V,Dn”というテストにおいて，分岐命令の使い方をまとめると次のようになる．

テスト	符号付き		符号なし	
	真で分岐	偽で分岐	真で分岐	偽で分岐
D_n>V	BGT	BLE	BHI	BLS
D_n≧V	BGE	BLT	BHS(BCC)	BLO(BCS)
D_n=V	BEQ	BNE	BEQ	BNE
D_n≦V	BLE	BGT	BLS	BHI
D_n<V	BLT	BGE	BLO(BCS)	BHS(BCC)

Vは適当なソース・オペランド

注4：　「7章のバスエラーとアドレスエラー」参照．

注5：　この目的には，“SUBA　An,An”を用いる．

注6：　すなわち条件が真・偽であるかに応じて＄FFまたは0となる．

注 7： この設定のもとではループを1回だけ通ってしまうので，正しくは次のようになる.

```
            BRA     L1
L2          .
            .
            .
            .
L1          DBRA    Dn,L2
```

注 8： ハードウェア構成によっては上位バイトに置くこともある.

注 9： 「8章の入出力」参照.

注10： 語長8ビット，パリティなし，送・受信割込み禁止，1/16モード.

注11： 実際にこの意味で〝非同期〟というわけではなく，シリアル・ライン上のビット・データの形式が〝非同期〟的に通信できるように考えられているため.

CHAPTER 4
スタックとサブルーチン

4.1	サブルーチン	99
4.2	アブソリュート・ジャンプ	106
4.3	実効アドレス	109
4.4	スタック領域の割付け	112
4.5	メモリ診断プログラム	116

はじめに

コンピュータ内部のプログラムで使用されるデータを組織化する上で，最も一般的な方法は，スタックの使用です．この技法は一枚の用紙の上に別の用紙を置いていくのと同じで，記憶させる個々の新しいデータを最後のデータの〝上〟に置いていきます．しかし，データを除去するためには，最も新しくスタック上に置いたデータから取り去らなければなりません．その後で，前のデータを除去したり，新しいデータを追加することができます．

スタックは，コンピュータ内で1つのメモリ領域として表され，この領域を高い番地から低い番地へ向かって使用します．このように，表記方法が〝上下逆〟になっているため，TOS(Top Of Stack：スタックの先頭)という表現をしますが，これは，現在使用されているスタック領域の最下位のメモリ番地を意味します．

初期的には，レジスタはスタック領域の最上位の番地を指すようにセットされます．ある値を記憶さる必要がある場合は，記憶したいオブジェクト(処理対象)の大きさだけポインタをデクリメントし，そのオブジェクトを更新済みのレジスタによって示されるメモリに置きます．また別の値を記憶する必要がある場合は，ポインタ(スタック・ポインタ)を更新し，最初に記憶したオブジェクトに隣接する位置に，そのオブジェクトを置き，まったく同じ操作をします．

スタックについて唯一の問題は，スタックにオブジェクトが記憶されたのと逆の順番でしか，それを除去できないという点です．そのため，スタック上の2番目のオブジェクトを除去するためには，逆のプロセスで，スタック・ポインタによって示される番地から情報を読み出し，ポインタにオブジェクトの大きさを加えます．このようにして，新しいオブジェクトをスタック上に置いたり，そこに最初に置いたオブジェクトを除去することができます．

68000では，8個のアドレス・レジスタが装備されており，そのすべてをスタック・ポインタとして使用することができます．プレデクリメントおよびポストインクリメントモードを使用できるので，この目的のためには，ただ単にレジスタを適切なTOSポインタの初期値にセットするだけでいいのです．その後で，このメモリ領域に結果を保存することができます．これは，レジスタの元

の内容を破壊せずに，いくつかのレジスタを計算で使用する必要がある場合に特に便利です．次に例を示します．

```
        MOVEA.L #$2000,A3     Set up A3 to stack top
* Set up data registers to important values
*       ...
        MOVE.L  D0,-(A3)      Save register D0 on stack
        MOVE.L  D1,-(A3)      Save register D1 on stack
        MOVE.L  #$123,D0      Use D0 and D1 in some way
*       ...
        MOVE.L  (A3)+,D1      Restore register D1
        MOVE.L  (A3)+,D0      Restore D0
*       ...                   Use old values of D0, D1
```

この例では，A3をスタック・ポインタとしてセットし，次にデータ・レジスタに必要なロング値をロードしていきます．プログラムの後半では，データ・レジスタ内のロング値が必要となりますが，他の計算で必要とされるレジスタを既に使い切っています．

一つの解決方法として，D0とD1に入っていた元の値を名前付きのメモリ番地に記憶し，終わったらそれらを復元する方法が考えられます．このような処理を必要とする状況がよくあるでしょうが，この場合には，スタックを使う方がはるかに簡単です．さらに重要な点としてスタックを使用することにより，ピュア・コードと位置独立なコードの両方を確実に書ける，ということがあります．この利点については，前述していますので説明は省略します．

したがって，スタック上にデータ・レジスタの現在の値を保存することにします．A3には，初期値として$2000が入っています．ロングのMOVE操作を行っているので，プレデクリメント・アドレッシングモードにより，A3から4が減算されることになります．D0の内容は，A3の新しい値によって示される番地に保存されます．言い換えると，バイト$1FFC～1FFFに記憶されます．次の命令は，D1の内容をバイト$1FF8～1FFBに保存します．そしてA3は値$1FF8を持って終わりとなります．

これでD0とD1を計算に使用できるようになりました．計算が終わったら，ポストインクリメント・アドレッシングモードを使って，A3によって示される番地からロードすることで，古い値を復元します．このようにして，D1の元の値は番地$1FF8から，そしてD0は$1FFCからロードされます．値がスタック

に記憶されたのと逆の順序で，スタックから値を取り出す点に注意してください．

MOVE 命令には特殊な形式があり，それはスタックを取り扱う場合に特に便利なものです．前述の例では，スタック上へ2つのレジスタの値を保存し，その処理のために2つの命令を使用しています．もし16個のレジスタの値を保存したい場合には，16個の命令を使用しなければなりません．この場合，32バイトのコードを消費し，相当な実行時間がかかります．

MOVEM 命令は，スタック上に値を保存することを支援するように設計されています．この命令は，1～16個のレジスタがスタックに保存されるか，あるいはスタックからロードされるかを指定します．またこの命令は，1つの引数としてレジスタのリストを取り，もう1つの引数として実効アドレスを取ります．レジスタのリストはワード値に変換されます．ここで1にセットされたビットは，対応するレジスタが移動操作の対象となることを示します．この形式は，4バイト長の命令を1つ使用することによって，16個のレジスタ全体をスタックとの間でやりとりできることを意味し，16個の別々の命令を使用するよりも，はるかに短時間で済みます．

前述の例を修正して，次のようにすることができます．

```
        MOVEA.L #$2000,A3     Set up A3 to stack top
* Set up data registers to important values
*       ...
        MOVEM.L D0-D1,-(A3)   Save registers D0 and D1
*                             on stack
        MOVE.L  #$123,D0      Use D0 and D1 in some way
*       ...
        MOVEM.L (A3)+,D0-D1   Restore registers D0, D1
*       ...                   Use old values of D0, D1
```

MOVEM 命令は，最初のレジスタ，ハイフン，最後のレジスタという形式で，レジスタのリストを取ります．最初のレジスタと最後のレジスタも含めて，その間にあるレジスタがすべて，スタックとの間でやりとりされます．

レジスタのリストのもう1つの形式は，レジスタ名とレジスタ名の間を，スラッシュで区切る方法です．この2つの形式を混合することもできます．

```
MOVEM.L D1-D4/D7/A0-A2/A6,-(A3)
```

これは，レジスタD1～D4, D7, A0～A2およびA6を保存します．アセンブラによっては，1つの範囲内("－"形式)でデータ・レジスタとアドレス・レジスタの混合した形式を認めないものもあります．したがって次の形式は，

```
MOVEM.L D0-D7/A0-A6,-(A7)
```

A7によって示されるスタック上に，A7を除くすべてのレジスタを保存する場合に必要です．

スタック上にレジスタが保存される順序は，アセンブラに対してレジスタ・リストを指定した順序とは無関係です．というのは，アセンブラはただ単に指定されたレジスタが転送に加わることを示すために，命令内の適切なビットをセットするだけだからです．レジスタが保存される順序は，A7からA0へ，そして次にD7からD0へです．これらが復元される順序は，この逆であり，D0がレジスタのリストに指定されている場合，D0が最初にロードされ，次にD1, …, A0, …, 最後にA7となります．

ここまでの例で値を保存する場合は，プレデクリメント・アドレスモードを，そしてそれらを復元する場合は，ポストインクリメント・アドレスモードを使用してきました．このモードを使用した場合，レジスタは示されたとおりに転送され，スタック・ポインタとして使用されるアドレス・レジスタは，転送されるレジスタの合計の大きさ分だけ，インクリメントまたはデクリメントされます．このように，MOVEM命令はレジスタを保存または復元するのに必要とされる，相当数のMOVE命令と，非常に類似した作用をします．ただし，多くの相違点があります．

第1に，MOVEMにはワードまたはロングの形式しかありません．この操作を使って，スタック上に単一のバイト値を保存することはできません．ワード値が復元された場合，スタックから読み取られたワードを符号拡張して得られた32ビット値が，レジスタの全体にロードされます．このことはつまり，レジスタの下位16ビットを保存し，その下位半分を演算で使用し，その後でそれらの値の上位16ビットを失わずに復元することはできない，ということを意味しま

す．したがって，一般的にMOVEM命令のロング形式を使用して，すべてのレジスタの全内容を保存するべきであると言えます．

第2に，MOVEM命令は技術用語でいう〝**プレフェッチ**〟を使用します．すなわち68000プロセッサは，可能な限り速くレジスタを転送しようとするので，このため，そのメモリ番地が実際に必要となる時点より少し前に，必要なメモリ番地を読み込んでしまいます．これによって，複数の転送を素早く行うことができますが，プロセッサがレジスタのリストの終わりに達した場合，メモリの1ワードを多く読み過ぎたことになります．スタック・ポインタの最終的な内容は正しいものなので，これは重要ではありません．そのため，メモリの1ワードが読み取られ，忘れられたとしても問題はありません．問題が生じる唯一の場合は，メモリの最上位番地でスタックが始まる場合です．

この場合，レジスタはメモリの先頭からの番地に記憶されますが，これらのレジスタが復元された場合，プロセッサはメモリの境界を越えて，もう1ワードを読み込もうとします．プロセッサはこうして得られた値を忘れてしまいますが，このアクセスは通常，バスエラーを引き起こし，プログラムは期待どおりの作用をしなくなります[注1]．

最後に，MOVE命令がコンディション・コードを変更するのに対し，MOVEM命令は変更しません．これによって結果の如何を示すためにサブルーチンでコンディション・コードをセットすることが可能となります．このコードは，レジスタの元の値が復元されても不変です．

前述したとおり，レジスタは，プレデクリメントまたはポストインクリメント・アドレスモードを使用した場合のみ，規定の形式で転送されるとしましたが，MOVEM命令は制御アドレッシングモードに属する他のアドレスモードでも使用することができます．転送がメモリに対して行われる場合，アドレスモードは制御可変アドレッシングでなければなりません．言い換えると，プログラム・カウンタ相対モードはメモリから読み出す場合でしか使用できません．

プレデクリメントまたはポストインクリメント以外のアドレスモードで使用した場合，転送の順序は常に同じです．この順序は，D0～D7，次にA0～A7で，ポストインクリメント・アドレスモードが使用された場合と同じです．

```
MOVEM.L D0-D7,$2000
```

これは，D0の内容をバイト＄2000～2003に，D1の内容を$2004～2007にという具合に記憶します．これらの値を再ロードするには，

```
MOVEM.L $2000,D0-D7
```

このように書く必要があります．

4.1 サブルーチン

これまでに示した**MOVEM**命令の例の中で，スタック・ポインタを保持するアドレス・レジスタとして，A7を使用している例があります．どのアドレス・レジスタでも，プレデクリメントまたはポストインクリメントモードでスタック・ポインタとして使用することができますが，A7を使用するのが一般的です．この理由は，他の多くの命令が，スタックとして使用されるメモリ領域を示すレジスタとして，A7を想定しているからです．レジスタA7には実際には2つの形式があります．それらはユーザー・スタック・ポインタ(**USP**)とスーパーバイザ・スタック・ポインタ(**SSP**)です．ここでは，A7が常にスーパーバイザ・スタック・ポインタを参照すると想定します．

68000用のプログラムを書く場合，実際の作業を開始する前に，スタック・ポインタをスタック領域の先頭に確実にセットしておくのが普通です．この処理は，プログラムを実行する機能を提供するオペレーティング・システムまたはモニタによって行われますが，常に明示的に行うことができます．例えば，次のようにです．

```
MOVEA.L #$2000,A7
```

これは＄2000より下の領域をスタックとしてセットします．スタック領域があふれないという保証はないので，一般的にスタックの成長を見越して余裕をとっておく必要があります(スタックの成長する様子については後述します)．そのために，スタックが＄2000から＄1000番地まで成長することができるよう

にします．すなわち，プログラムが＄2000番地から開始できる，ということを意味します．MOVEM 命令の項で説明したように，スタック・ポインタを使用する前にスタック・ポインタを変化させるプレデクリメント・アドレッシングモードを使用しているので，スタック・ポインタの初期値によって示されるバイトは，実際には書き込まれない，という点に注意してください．

スタック・ポインタのセットが済んだら，命令を実行してみることです．それは，A7 が有効なスタック領域を指していることを前提とするものです．たぶん，最もわかりやすい例は，サブルーチンへの分岐(BSR)命令でしょう．これは非常に重要な命令で，BRA と同じ方法でプログラムの他の領域へジャンプできるものです．実際に多くの点で BRA と同じですが，BRA を使って，どこかへジャンプしたい場合，どこからジャンプしてきたかを確認する方法はありません．

プログラム内で，次の命令を使用する状況が多くあります．

```
BRA        ERROR
```

ラベル ERROR の位置では，何らかのエラーメッセージを表示して停止することができます．ERROR の位置のコードでは，分岐の行われた位置を見つける方法はありません．レジスタまたはメモリ内の特定の番地に値を入れ，ERROR へジャンプを行った理由を示すことによって，適切なメッセージを表示することができます．このメッセージを表示したら，また，プログラムの実行を続けたいとします．この場合，BRA のかわりに BSR 命令を使用します．プロセッサの動作は，まず，次の命令の番地を保存し，次にラベル ERROR への分岐を行います．コード ERROR では，処理の終わったら，この保存された番地を使って戻ることができます．したがって，メッセージを書き出したら，BSR の直後の命令から実行を続けることができます．さらに，この保存されている番地を検査して，エラーが発生した正確な位置を特定するのに使用することができます．

BSR によって保存された値のことを**戻り番地**といい，これが A7 によって示されるスタックに保存されるということは，すでにおわかりのことと思います．もし次の命令の番地を PC というレジスタにロードできたとすると，BSR の動作は，次の動作と類似したものとなります．

```
MOVE.L  PC,-(A7)      Save return addr on stack
BRA     ERROR         Branch to subroutine
```

実際にはプログラム・カウンタの値を明示的に参照する方法は有りません．BSR命令はA7をデクリメントし，この新しい値によって参照される4バイトに戻り番地を記憶し，次にラベルへ分岐するという動作を一度に行います．

ここで，スタックに保存されているこの値を取り出し，BSRの後の命令にジャンプして戻りたいとします．もし必要ならば，この値はスタックの先頭から明示的に読み取ることができます．そのため，次のようなコードを使って戻ることができます．

```
MOVEA.L (A7)+,A6      Extract return address
JMP     (A6)          And jump to that location
```

戻るための一般的な手段はRTS(サブルーチンからのリターン)の使用です．これはBSRとまったく逆で,スタックに保存された値から戻り番地を読み取り,A7を4だけインクリメントして,そのスタック・スロットを再び使用可能にし,次にこの新しい番地にジャンプします．

これでスタックを使用することの利点が，おわかりいただけたと思います．BSRは必要なら何回でも使用することができ,また,他のBSRによって入ってきたコード・セクション内でも，BSRを使用することができます．BSRを実行するたびに，A7の値は4だけデクリメントされ，スタックの次のスロットに新しい戻り番地が保存されます．RTSを検出するたびに，スタックはインクリメントされ,サブルーチンに入るのに使用されたBSRの直後の命令から実行が続けられます．

A7のこの特殊な使用法を理解し，A7が常に確実に適切なスタック領域を指定するようにしなければなりません.ある種のアセンブラではレジスタA7としてSP(スタック・ポインタの意)という表記を用意しており，これを使用することによって，ユーザーに普通のアドレス・レジスタではないことを意識させるようにしています．もちろん，多くの状況下で，A7を他のアドレス・レジスタと同様に使用することができますが，重要な相違点が一つあります．

スタックに記憶された値は，偶数番地に揃えなければならず，ハードウェア

がこのことを保証しています．すなわち，レジスタA7を指定してバイト・サイズの命令をプレデクリメント，またはポストインクリメントモードで使用した場合，A7の値は他のレジスタの場合のように1ではなく，2だけ変化します．

サブルーチンの概念は非常に重要なものであり，プログラミングの経験のある人なら，誰でも知っているはずのものです．この概念とは，何かの動作を行うために同じコード・セクションを何回も書くかわりに，それを1回だけ書いてサブルーチンとして使用することです．最もわかりやすい例は，第3章で説明したような1文字出力ルーチンでしょう．文字を書きたいときに，文字を出力ポートに入れる前にACIAがレディ状態になっているかどうかを調べるテスト・コードを常に埋め込む必要があるとしたら，プログラム空間が非常に無駄になります．そのかわりに，この処理を行うサブルーチンを書き，文字を書き出すにはBSRを使ってサブルーチンを呼び出し，文字を書き終えたら，RTSによって元のコード位置へ戻ることにします．

```
WRCH      BTST      #1,A_CTRL
          BEQ.S     WRCH
          MOVE.B    D0,A_DATA
          RTS
```

このサブルーチンはシリアル・ポートに接続された装置(すでに正しくセットされていると想定)に文字を書き出します．このサブルーチンを使用するためには，まずレジスタD0の下位バイトにその文字を書き込んでおき，ラベルWRCHを呼ぶBSRを組み入れるだけです．そうするとサブルーチンはまずACIAがビジー状態がどうかを調べます．もしビジー状態ならレディ状態になるまで何度もループを繰り返します．その後D0で渡されている文字は，ACIAのデータ・レジスタに入れられ，WRCHを呼び出したBSRの次の命令に制御が移ります．

一般的に，プログラマはすぐ使える有用なサブルーチンを数多く持っています．たいていのプログラムは類似した点を持っていて，例えば，ほとんどのプログラムは，何かの結果を書き出す必要があります．そのため，プログラムを可能な限りサブルーチンに分割するのが普通で，便利なサブルーチンを保存しておいて，後で他のプログラムに組み入れられるようにします．一般的に，サブルーチン・ライブラリには，非常に多くの異なるサブルーチンが含まれています．単純に文字を書き出しながら，単に出力を行う場合に必要とされる，多

くの便利な操作があります．例えば，文字列や10進数，または16進数などを書き出したい場合です．

このサブルーチン・ライブラリは，一度に少しずつ作成することができますが，従うべきプログラミング上の規則があります．これはどんな言語またはどんなプロセッサを使用するにせよ，従う必要のあるものです．しかし，幸いにも68000の命令セットには，よく構造化されたプログラムの作成を支援する機能があります．

第1の規則として，すべてのサブルーチンをあらゆる環境で確実に使用できるようにすることです．例えば，サブルーチン・ライブラリを拡張して，文字列を書き出すためのサブルーチンを追加する場合を考えてみましょう．アドレス・レジスタA4で文字列を指して，BSRを使ってこのサブルーチンを呼び出します．文字列とは，バイト値0で終わる文字の連なりであると定義します．次のように書きます．

```
WRITES  MOVE.B  (A4)+,D0     Load byte from string
        BEQ.S   WOVER        Branch if end of string
        BSR.S   WRCH         Write out character
        BRA.S   WRITES       Back to next character
WOVER   RTS
```

ここで文字列から1バイトを取り出し，A4をインクリメントして次回に使用できるようにします．このバイトが0である場合，ラベルWOVERに分岐し，呼び出し元に戻ります．それ以外の場合は，ライブラリ内のサブルーチンを使って文字を書き出し，始めに戻って文字列から次の文字を取り出します．ここで，サブルーチンへの分岐命令のショート形式をBSR .Sと指定している点に注意してください．これはBRA .Sと同様であり，ラベルが128バイト未満の距離にある場合に使用できる，より短い命令形式です．

このルーチンは，必要な役割を完全に果たしますが，ただし多くの問題があります．最初の問題点は，サブルーチンWRITESを使用した場合，D0が破壊されることを覚えておかなければならない点です．実際に，下位バイトは常に0にセットされます．このアプリケーションでは，レジスタD0に何が起こっても何の処置も取りませんが，このサブルーチンをあらゆる状況で使用できる便利なサブルーチンにするためには，レジスタが破壊されるということは，大変な

欠陥です.

コードをさらに注意して見ていくと，アドレス・レジスタA4も破壊されることがわかります．このレジスタは，文字列の終わりを越えたバイトを指すようにセットされます．このサブルーチンのユーザーは，自分の文字列がA4に渡されることを知っていなければなりませんが，ユーザーが要求した文字列出力の処理の副作用として，A4の値を変化させてしまうことは，事実上フェアなことではありません．一般的かつ有効な規則として，サブルーチンによってレジスタを変更しないこと(ただし，サブルーチンの結果がレジスタに戻される場合はもちろん例外とする)が挙げられます．

レジスタを変化させるのはプログラミングのスタイルとして良くない，と言いましたが，これから使用するレジスタの値をどこに退避するのかを決定しなければなりません．1つの方法として，メモリ内の1つの領域を単純に割り付け，それを記憶域として使用する方法があります．これで解決したように感じられますが，多くの問題点があります．第1に，個々のサブルーチンに対して別々の記憶域を使用しなければなりません．さもないと，1つのサブルーチンがほかを呼び出した場合，退避された値が，レジスタの内容を退避しようとする他のサブルーチンによって，重ね書きされる可能性があります．この場合，空間が無駄になり，また組織化するのも困難です．第2に，メイン・プログラムで各サブルーチンに対して割り付けた記憶域を，他の目的(例えば，書き込み)のために使用していないことを確認しなければなりません．第3に，プログラムはリエントラント(詳細は第2章を参照)になりません．

正解はもちろんMOVEMを適切な位置で使用して，レジスタ値をスタックに退避することです．BSRを使用するために，レジスタがメモリ内の1つの領域を指すようにすでに割り付けてあります．この領域に充分な大きさがある限りすべてうまくいきます．つまり，サブルーチンのネスティングが最も深くなった時点でも，すべての退避されたレジスタと戻り番地のための領域が充分確保できるならば，OKだということです．この方法では，実際に必要な場合にしか空間を使用しないので，これは効率の良い空間の使用法だと言えます．レジスタを保存する必要のあるサブルーチンはたくさんありますが，最大のスタック消費は，単に各々のサブルーチンが呼び出す，サブルーチンの最大個数に応じた量でおさえられます．

文字列を書き出すサブルーチンを次のように改良してみました.

```
WRITES   MOVEM.L D0/A4,-(SP)   Save registers
WR1      MOVE.B  (A4)+,D0      Extract byte
         BEQ.S   WR2           Branch if end
         BSR.S   WRCH          Write character
         BRA.S   WR1           Get next character
WR2      MOVEM.L (SP)+,D0/A4   Restore registers
         RTS
```

これでWRITESをどこでも必要なところで使用できるようになり，レジスタが変化する心配はなくなりました．実際にこのルーチンで使われるスタックを例外として，メモリは変更されません．この領域は，サブルーチン・コールを用いると常に変更され得る部分です(つまり，戻り番地をしまう領域と呼び出されたサブルーチンが使うスタック領域のこと)．

汎用サブルーチンを記述する場合に頭に入れて置かなければならない規則として，このほかに2つあります．

その1つは，サブルーチンに対して渡される引数とそこから戻される結果を入れるために，一貫性のあるレジスタ群を使用しなければならないということです．そうすれば，ユーザーは，引数がレジスタD1, D2などに入り，単一の結果がD0で戻されるといつでも考えるでしょう．明らかにこれがいつでも可能であるとは限りません．というのは，ルーチンによっては引数がデータ・レジスタに入っているとみなすものもあれば，アドレス・レジスタに入っているとみなすものもあります．しかし，このこと(一貫性のあるレジスタ群を使用すること)は，一般規則として有効です．なぜなら特定のサブルーチンを呼び出す前に，どのレジスタにどの値が入っているべきか，プログラマが混乱しないですむようになります．

もう1つはサブルーチンの出口は一般的に1個であるということを保証することが重要です．1つのサブルーチンの中に複数のRTS命令を指定するよりも，1個だけにしておいて，必要な場合にその番地へ他の番地から分岐するようにした方が良いと言えます．これはすなわち，レジスタの復帰，スタックの割付け・解放などの処理が，すべて1つの場所で行えるということを意味します．

この場合，余分なレジスタを使用および保存するためにサブルーチンを変更する場合，単一の入口および出口のコードだけを変更すればよいのです．スタ

ックにレジスタを保存して，RTS を実行する前にそれを復帰するのを忘れてしまうことは，非常によくあるプログラミングエラーです．プログラムは，スタックに保存されているレジスタの値によって指定される番地へジャンプするので，どこを参照するとも限らず，その作用は致命的です．これは特に見つけにくいエラーです．というのは，デバッグ情報が新しいプログラム・カウンタを参照するものであるのに対し，必要な情報は，プログラムがなぜその最初の位置にあったのかということだからです．

4.2 アブソリュート・ジャンプ

ループのセットを行う BRA，サブルーチンを呼び出す BSR や BEQ などの条件付き分岐について既に説明してきたように，プログラムの他の部分へジャンプできることは，非常に重要なことです．

これらの場合，すべてに通じて，命令のことをジャンプではなく，分岐と呼んできましたが，これには理由があります．これらすべての分岐命令では，現在位置に対する相対的な番地へ，プログラム制御を移すように指定しています．次のような文を書くことはできます．

```
        BRA      LOOP
```

しかし，アセンブラはこの文をプログラム・カウンタの現在値とラベル LOOP の番地との差を埋め込んだ命令に変換します．このオフセットはジャンプが後方向と前方向のどちらであるかに応じて，負の場合と正の場合があります．このオフセットはワードで指定することができ，この場合オフセットを入れるために16ビットを使用します．または，命令の短形式を使用する場合は，バイトで指定することができます．

BRA の代わりに JMP を使用できますが，これはアブソリュート・ジャンプを表します．JMP 命令の一部分として使用されるアドレスは実アドレスであり，命令が実行されるとそこへ制御が移ります．

```
        JMP     LOOP
```

この命令の作用は，BRA 命令と似ていますが，多くの相違点があります．

第1に，JMP 命令では完全なアドレスを指定するので，ラベルへジャンプする場合は，より長くなります．

第2に，そのセクション自体の中のラベルへジャンプする JMP を使用するコード・セクションは，位置独立ではあり得ません．68000では，位置独立な方法でコードを書くことのできる命令を数多く用意しています．例えば，BRA 命令は，現在位置から24バイト離れた番地へのジャンプを指定することができます．これは，プログラムがメモリのどこに入っていても作用します．

プログラムが JMP 命令を使用する場合，その値は，任意の制御アドレッシングモードを使って指定することができます．このときアドレスをアブソリュート値として指定した場合，そのアドレスは，単一の，特定のメモリ番地を参照します．このプログラムは，先頭の ORG 文で指定された番地のメモリにロードされた場合にのみ作動します．

多くの場合において，特定番地のメモリにプログラムをロードすることは，完全に許されますが，コンピュータがオペレーティング・システムを走らせているときには，不可能な場合もあります．ここでは，プログラムは当然のことながら，任意の使用可能な空間にロードされます．そして，そのプログラムが位置独立な方法で作られていれば変更なしで動作します．さもなければ，プログラムに何らかのリロケーション情報を含んでいなければなりません．この情報は，リロケーションを行うオペレーティング・システムを前提とする或る種のアセンブラで生成されます[注2]．このことにより，プログラムがロードされたアドレスで走ることが保証されます．不都合なことにすべてのアセンブラが適切なリロケーション情報を生成するわけではありません．したがって，位置独立なコードを書くのは，良い習慣であると言えます．

位置独立なアセンブリ・コードの利点として，メモリ内の任意の位置にプログラムをロードできるばかりでなく，必要ならばいろいろな位置へ移動することも可能です．プログラムの実行中には，よほど多くの注意を払わない限り，このようなコードの切混ぜ(shuffling)は起こりません．というのは，スタック

にメモリ内のアブソリュートな番地を参照する戻り番地が入っているからです．しかし，プログラムの実行と実行の間では，確かに起こる可能性があります．

JMP 命令を位置独立なプログラムで効果的に使用することができます．そのような JMP 命令の使用は，ユーザー独自のコード・セクション内のラベルに対して行った場合のみ，位置独立ではなくなります．JMP 命令の一般的な使用法として，一定のメモリ番地に常駐していることが知られているプログラム・セクションへジャンプする場合があります．例えば，EPROM のある番地にモニタが存在しており，その中にウォームスタート・エントリ・アドレスがあり，ユーザーのプログラムが終了したら，そこへ入っていかなければならないとします．この場合，プログラムの終わりは，次のように書きます．

```
        JMP     WARMS
```

これにより，ユーザーのプログラムがどこにロードされたとしても，確実にモニタに入るようになります．

BSR 命令にもまた，JSR というよく似た命令があります．やはり JSR もオフセットではなくアドレスを取り，特定のメモリ番地に存在していることが知られているサブルーチンを呼び出すのにも使用できます．

JMP と JSR 命令は，特にアドレスとして任意の制御アドレッシングモードを取るという理由から非常に重要です．上の例で，一定のメモリ番地へジャンプするために，アブソリュート・アドレッシングモードを使用しました．また，位置の独立性を保ちながら，プログラム内の番地を参照するために，プログラム・カウンタ相対モードを使用することもできます．この場合 BRA または BSR と非常によく似た作用となります．

ここで使用できる最も便利なアドレッシングモードは，インデックスモードの一つでしょう．1文字のコマンドで選択されるプログラムを書く場合を考えてみましょう．ターミナルでコマンドが打ち込まれるたびに，サブルーチンが呼び出されてそのコマンドを実行します．これをプログラムするには，個々の ASCII キャラクタに対して4バイトを持ったテーブルを作成します．テーブルの個々のエントリは，そのキャラクタが打ち込まれたときに呼び出すルーチンのアドレスを表しています．したがって，テーブルの最初のエントリはコード

0(ASCII キャラクタの NUL に対応する)が打ち込まれたときに呼び出されるルーチンです．"A"が打ち込まれたときに呼び出されるルーチンは，キャラクタ"A"に対するオフセット，すなわちロングワード・オフセット＄41にあります[注3]．

```
ASL.L    #2,D0          Multiply by four
MOVEA.L #TABLE,A3       Get table address
JSR      0(A3,D0.W)     Call subroutine to do job
```

1行目は，レジスタ D0 の内容を左へ2個分シフトすることによって，それを4倍します．これは，テーブルの個々のスロットが4バイトを使うため必要です．2行目は，テーブルのアドレスを取り，それをアドレス・レジスタ A3 にロードします．3行目は，A3 と D0 の和によって示される番地に記憶されているアドレスを取り出し，スタックに適切な戻り番地を置き，それをサブルーチンとして呼び出します．

4.3 実効アドレス

前の例では，MOVEA 命令をイミディエイト・アドレッシングモードで使用して，TABLE の値を A3 にロードしています．これは完全に正しく動作しますが，命令は位置独立ではありません．実際の動作は，68000に対しデータ値をレジスタにロードするよう指示しているのです．このデータ値は，テーブルの先頭で宣言されたラベルの値です．プログラムが ORG 文で宣言されたメモリ番地にロードされない場合，MOVEA 命令は，もうそのデータ値がテーブルの先頭を参照しないのにもかかわらず，なおこのデータ値をロードします．

解決方法は，LEA(Load Effective Address，実効アドレスのロード)命令を使用することです．この命令は，デスティネーションとしてアドレス・レジスタを指定する場合にのみ使用することができ，ソースとして与えられたアドレスを，例えば，MOVE 命令と同じ方法で評価します．その後，そのアドレスの内容をロードする代わりに，アドレス自体をデスティネーション・レジスタに入れます．

次のプログラム・セグメントについて考えてみましょう.

```
        ORG      $1000
LAB     DC.L     1234
        MOVEA.L  LAB,A3
        MOVEA.L  #LAB,A3
        LEA.L    LAB,A3
```

ここで＄1000番地に4バイトを宣言し，それを1234に初期設定します．最初のMOVEA命令はLABによって与えられたアドレス,すなわち＄1000を評価し，その後で，番地の内容，すなわち1234をロードします．2番目のMOVEAは，レジスタA3にラベルLABで与えられるイミディエイト値，すなわち＄1000をロードします．ただし，この命令は位置依存であり，プログラムがアブソリュートモードでアセンブルされた場合にのみ動作します．

この処理を行うさらに良い方法は，LEAを使ってアドレスをロードすることです．これは，もし必要ならプログラム・カウンタ相対アドレッシングを使って，ラベルLABによって与えられるアドレスを評価します．そのため，コードは位置独立となります．LEAはアドレスを評価し，アドレス自体を指定されたレジスタに入れるということを思い出してください．そのアドレスに記憶されている値はアクセスしません．

当然のことながら，この命令はロング形式でのみ意味を持ち，LEAという形式は，LEA.Lと同じです.命令はそれぞれデフォルトの長さが異なるので,個々の命令の長さを常に明示的に指定する習慣をつけたほうがいいと思われます[注4].

前の例で，ラベルLABに対する参照は，プログラム・カウンタ相対アドレッシングを使って行われており，命令が位置独立であることが保証されるので，その意味でLEA命令は重要です．

LEAは，プログラム内の位置のアドレスをロードするために常に使用すべきであり，これに対しMOVEのイミディエイト形式は，イミディエイト・データ値をロードするために使用すべきです．

さらにLEAを使って，アドレス評価の一部分として単純な加算を行うことができます．

例えば,

```
        LEA     20(A3),A3
```

これは,20(A3)によって指定される実効アドレスを評価します.これは,A3の内容に定数20を加えたものであり,結果はA3に入れられます.すなわちA3に20を加算したのと同じ結果になります.任意の制御アドレッシングモードを使用することができるため,次のように指定することもできます.

```
        LEA     20(A2,D1.L),A3
```

これは定数20,A2の内容およびD1の内容の和をA3にロードします.アドレスは通常24ビット長にすぎませんが,アドレス・レジスタの全32ビットがこの方法で変えられます.

LEAと類似の命令に,PEA(Push Effective Address,実効アドレスのプッシュ)があります.この命令は,ソースとして指定されたアドレスをLEAと同じ方法で評価しますが,結果として得られた実効アドレスをアドレス・レジスタに入れるのではなく,それをスタックに記憶します.実際には,この動作はBSRおよびJSRの一部,つまり次の命令の実効アドレスがスタックに記憶されるといった動作と同様にして行われます.BSRの作用を次のようにシミュレートすることができます.

```
        PEA.L   NEXT            Save return addr on stack
        BRA     SUBR            Branch to subroutine
NEXT    ...     Return to here
```

ここで,戻り番地をスタックにプッシュし,次にBSRではなくBRAを使ってサブルーチンに入ります.サブルーチンがRTSによって戻るとき,スタックに保存されている戻り番地を取り出し,NEXTでラベル付けされた命令に戻りますが,この場合,この命令はサブルーチン・コールの次の命令となっています.ここではBSRの動作をシミュレートしましたが,もちろんPEAに対して指定したアドレスは,BRAのあとの命令を参照する必要はなく,どこでも参照する

ことができます.

さらにPEAを使って単純な加算を行うことができます. 例えば,

```
PEA.L    20(A3)       Save A3 + 20 on stack
MOVE.L   (SP)+,D0     Load D0 with saved value
```

これは"A3の内容+20"をスタックに保存し, この値をスタックから取ってD0に読み込みます. これはデータ・レジスタをデスティネーションとして使用し, LEA命令と同じ作用を得るための1つの可能な方法です.

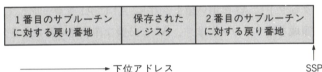
4.4 スタック領域の割付け

ここまでの例で, スタックを使って戻り番地や, 退避したレジスタのコピーおよび一時的な結果を収容する方法についてみてきました. しかし, いずれの場合においても, スタックから値を取り出す場合, 入れたときと逆の順序で取り出さなければなりません. 結果を保存したメモリ領域を割り付けることができ, いつでもこれらの番地を読み書きできればさらに便利でしょう.

この処理を行うための1つの方法が, 絶対番地を使用することですが, やはり位置独立の問題につきあたります. プログラムが使用するための特定のメモリ領域を確保しなければなりません. プログラムをメモリの任意の位置に置けるような処置をしたとしても, データ領域が特定の番地に結び付けられているなら, 何にもなりません. プログラム空間内にメモリ領域を確保し, プログラム・カウンタ相対アドレッシングを使ってこれらを参照することができます. ただし, ユーザーは, この方法で指定されたメモリ番地を読むことしかできません. というのは, 68000のアーキテクチャは, 非常に正確に, プログラム自身を書き換えるといった互いに重ね書きしあうプログラムの作成を禁止しているからです. もう1つの方法は, アドレス・レジスタを使用することで, このアドレス・レジスタが結果を入れるためのメモリ領域を確実に指すようにすることです. このアドレス・レジスタからのオフセットを使って, ユーザーのデータを記憶するために使用します. したがって, A1がユーザーのデータ空間を指している場合, 次のようにして位置を参照することができます.

```
DATA1    EQU      0              Data area offsets
DATA2    EQU      4
*        ...      Set up A1
         MOVE.L   #20,D1         Get value
         MOVE.L   D1,DATA1(A1)   Save in data area
*        ...
```

このプログラムはかなりうまく動作しますが，２つ問題があります．１つはデータ空間の割付けに関連する問題です．いくらかのフリースペースを入手して，A1がそれを示すように初期設定するには，オペレーティング・システム・コールを使用しなければなりません．第６章に適切なフリーエリア割付けパッケージの例を示しますが，このようなオペレーティング・システム・コールは，どちらかといえば高価です[注5]．２番目の問題は，データ領域を必要とするサブルーチンに入るたびに，この空間を割り付けなければならないということであり，そのため，どのサブルーチンも他のサブルーチンを呼び出すことはできません．すべての空間を食い尽くしてしまうのを防止するために，サブルーチンから出るときは必ず空間を確実に解放しなければなりません．

解決方法は，一時的(temporary)データに対して必要な空間をスタックから取ることです．これまでの段階では，スタックにデータを実際に入れる場合に，スタックが成長できるようにすることだけを考慮してきました．この前提に立って，１つのレジスタを保存し，次に他のサブルーチンを呼び出す，あるサブルーチンを考えます．スタックには，２番目のサブルーチンに対する戻り番地が入り，次に保存されたレジスタおよび１番目のサブルーチンからの戻り番地が入ります．

ではスタックの一部分をデータ領域として割り付ける方法について考えてみましょう．1番目のサブルーチンに入るとき，すぐにユーザーのレジスタを退避するので，スタックには退避されたレジスタ値と戻り番地が入ります．ここでデータ領域ポインタA1を，スタック・ポインタA7と同じになるようセットし，A1以降にユーザーの必要に見合った充分な空間が割り付けられるようにA7を変更します．2番目のサブルーチンを呼び出すとき，戻り番地はA7によって示されるスタック位置，ユーザーのデータ領域の次に記憶されます．

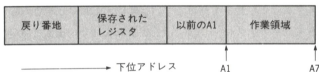
この段階で2番目のサブルーチンは自由にレジスタを保存することができ，そのサブルーチン自身のデータ領域を必要ならばスタックに割り付けることができます．このサブルーチンは，新しい作業領域を割り付ける前に，レジスタA1の元の内容を保存しなければならず，サブルーチンが終了したら，スタックを解放し，全レジスタを復帰しなければなりません．

この処理は複雑そうに思えるかもしれませんが，実際には非常に簡単であり，68000ではそのための特殊な命令も準備しています．これらの命令について説明する前に，サブルーチンに入るときとそこから出るときに，どんな処理をしなければならないかについて，復習しておきましょう．次の2つのコード・セクションは，エントリおよびエグジット手続きといいます．

▶エントリ手続き

1．サブルーチンに入るとき，A7は戻り番地を指し，A1は前のデータ領域を指しています．
2．使用する作業レジスタを保存します．A7は保存されたレジスタと戻り番地を指し，A1は以前のデータ領域を指しています．
3．スタック上にA1の旧い値を保存し，A1にA7をロードして，新しい作業領域を指すようにします．A7を必要な作業領域の大きさ分だけデクリメントします(スタックは低位アドレスに向かって延びることに注意してください)．

この結果，スタック・フレームは次のようになります．

戻り番地	保存された レジスタ	以前のA1	作業領域

▶**エグジット手続き**

1. A7 に A1 の値をロードし，作業領域の割付けを解放します．この段階で A7 は，保存された A1 の値，作業レジスタおよび戻り番地を指します．スタックから A1 の前の値を再ロードします．
2. スタックから，保存された作業レジスタの値を復元します．この段階で A7 は，戻り番地を指し，他のすべてのレジスタには，元の値が復元されます．
3. スタックから戻り番地をロードし，そこへジャンプします．

これを68000のアセンブリ・コードに変換してみましょう．エントリ手続きの段階 1 は，**BSR** または **JSR** を使ってサブルーチンを呼び出すことによって行います．段階 2 は，スタックに対してプレデクリメント・アドレスモードを使って，**MOVE**，またはもっと一般的に **MOVEM** によって行います．段階 3 は，**LINK** 命令によって 1 回の操作で行います．この命令は，ソースとして指定された，アドレス・レジスタをスタックに退避し，次に A7 の(更新済みの)値をそれにロードします．最後に，デスティネーションとして与えられたイミディエイト値を，スタック・ポインタ A7 に加算します．スタックは低位アドレスに向かって延びるので，オフセットには負の値を使用しなければなりません．

サブルーチンでは，データをアクセスするのに A1 からの負のオフセットを使います．このオフセットは，現在の A7 の値を越えてはなりません．このため，**LINK** 命令で必要十分なオフセットを指定することが大切です．スタックの先頭を越えたオフセットを使用した場合，そのオフセットによって示されるデータは，以降のサブルーチン・コールによって破壊されます．

同様にエグジット手続きも簡単です.段階1は,LINKの逆であるUNLKによって行います.スタック・ポインタA7は,UNLK命令の引数で指定されたレジスタからロードされ,次にこのレジスタはスタックの先頭からロードされます.段階2は,スタック・ポインタからの,ポストインクリメント・アドレッシングモードを使用したエントリ手続きと対応したMOVEまたはMOVEMです.最後に段階3は,単なるRTSです.

```
* Standard entry sequence
        MOVEM.L D0-D7/A0,-(SP) Save work registers
        LINK    A1,#-32        Allocate 8 long words
*       ...
* Perform work, using -4(A1) to -32(A1) for data
* Possibly call other subroutines
*       ...
* Standard exit sequence
        UNLK    A1             Deallocate workspace
        MOVEM.L (SP)+,D0-D7/A0 Restore registers
        RTS                    Return to caller
```

このような手続きは高級言語の開発者によってよく使用されているもので,PascalやAdaなど,スタックを使って動作するものには特によく見られます.高級言語から呼び出し可能なサブルーチンを作成したい場合は,その特定の言語のサブルーチンによって使用される標準形式に適合するようにしなければなりません.当然のことながら,異なった実装で実際に使われている方法によって,呼び出しの形式にはある程度の差異があります[注6].

4.5 メモリ診断プログラム

これまでの説明から,68000の完全なプログラムを充分書けるようになっています.これから紹介するプログラムは,それほどすごいものではありませんが,これまでに説明したいくつかの命令が入っています.

このプログラムは,一定の範囲のメモリ番地が,実際に期待どおりの動作をするかどうか,言い換えると,メモリが正しく働いているかどうかをチェックします.ハードウェアの構成上,メモリ障害は,一定の記憶装置に特定のビット位置が常に0または1になるという形で,しばしば発生します.単にメモリ

に0を書き込んで，これが常に同じ状態になっているかどうかをチェックするだけでは，充分とは言えません．というのは，このようなチェックでは，常に0を戻すビットを見つけ出すことができないからです．障害によっては，実際に特定のパターンを問題の番地に書き込んだ場合にしか明らかにならないものもあります．そのため，可能なすべてのビットの組合せを使って，メモリに対して徹底的なチェックを行う必要があります．

プログラムの最初の部分は，ターミナルへの情報の出力を取り扱います．サブルーチン WRCH は，レジスタ D0 に記憶された文字を書き出します．

```
A_CTRL   EQU       $840021         ACIA control port
A_DATA   EQU       $840023         ACIA data port

WRCH     BTST      #1,A_CTRL       Test for port ready
         BEQ.S     WRCH            Loop until it is
         MOVE.B    D0,A_DATA       Transmit character
         RTS
```

ここで,文字列を書き出すサブルーチン WRITES を定義することができます．レジスタ A1 によって示される文字列は，バイト値0によって終結します．

```
WRITES   MOVEM.L D0/A1,-(SP)   Save registers
WRS1     MOVE.B  (A1)+,D0      Extract character
         BEQ.S   WRS2          Zero byte - exit
         BSR.S   WRCH          Write character
         BRA.S   WRS1          Get next character
WRS2     MOVEM.L (SP)+,D0/A1   Restore registers
         RTS                   And return
```

個々の可能なビット・パターンについて，指定された範囲内のすべてのメモリ番地に対し，その値を書きます．この書込みが終わったら，再びメモリを通し読みして，値が変わっていないかをチェックしなければなりません．1つの番地へ書き込むことにより，他の番地が変化してしまう場合があるので，書込みの直後にチェックを行うのは，充分ではありません．ただし，書込みループとチェックループは同様なループであり，その差異は各ループで行われる処理だけです．

ややスマートな形でこの処理を行う方法として，メモリを通し読みするサブルーチンを使用する方法があります．このサブルーチンは，各番地について，

必要な処理を行う他のサブルーチンを呼び出します．1番目のサブルーチンはSCANという名前で，レジスタA2に，必要な処理を行う別のサブルーチンの番地が入っています．レジスタD0には現在のテスト・ビット・パターンが入っており，A0はテスト中の番地を指しています．これらのレジスタは両方ともA2によってアドレスされる2個のサブルーチンによって使用されます．そしてA0は，サブルーチンが呼び出されるたびに1ずつインクリメントされます．

```
SCAN    MOVE.L  A0,-(SP)       Save A0
        MOVEA.L #MEMLO,A0      Start of test area
SCN1    JSR     (A2)           Call routine to do work
* A0 is incremented by subroutine called
        CMPA.L  #MEMHI,A0      Check if loop finished
        BNE.S   SCN1           No .. carry on
        MOVE.L  (SP)+,A0       Restore A0
        RTS                    And return
```

SCANによって呼び出される2つのサブルーチンは簡単です．1番目のサブルーチンはD0に入っている値をA0によって指される番地に入れ，そしてA0をインクリメントします．

```
WRITE   MOVE.B  D0,(A0)+       Store value
        RTS
```

2番目のサブルーチンはD0に入っている値がA0によって指されるメモリ番地の内容と同じであることをチェックします．次からA0をインクリメントし，メモリの値が期待通りのものでなかった場合はメッセージを表示します．

```
READ    CMP.B   (A0)+,D0       Check memory
        BEQ.S   RD1            Same, return
* Memory not the same, so we must write error message
        MOVE.L  A1,-(SP)       Save previous A1
        LEA.L   MESS3,A1       Point to error message
        BSR.S   WRITES         And write it out
        MOVE.L  (SP)+,A1       Restore old A1
RD1     RTS
```

最終段階は，プログラムのメイン部分を書くことです．モニタまたはオペレーティング・システムによって，すでにACIAがプログラム用に初期設定されているものと仮定します．プログラムの実行中，現在のテスト値をレジスタD0に入れておきます．この値は＄FFに初期設定され，DBccを使って，すべての組合せ可能なビット・パターンをテストするループを制御します．DBccは，レジスタの下位16ビット全体をデクリメントするので，この初期設定を実行するために，ワード長の命令を使用します．また，A1がメッセージを指すようにしておき，ループが一回りするたびに，プログラムが確かに動作していることを示すようにします．

```
MEMLO   EQU     $4000           Start memory address
MEMHI   EQU     $8000           End memory address
CR      EQU     $0D             ASCII return
LF      EQU     $0A             ASCII line feed

MCHECK  LEA.L   MESS1,A1        Get initial message
        BSR.S   WRITES          Write message
        LEA.L   MESS2,A1        Get progress message
        MOVE.W  #$FF,D0         Set up initial value for
*                               test
* Start of loop changing test pattern
LOOP    BSR.S   WRITES          Write progress message
        LEA.L   WRITE,A2        Point to WRITE subroutine
        BSR.S   SCAN            Scan memory performing
*                               WRITE
        LEA.L   READ,A2         Point to READ subroutine
        BSR.S   SCAN            Scan memory performing
*                               READ
        DBRA    D0,LOOP         Decrement test pattern
* Test complete. Write another message
        LEA.L   MESS4,A1        Point to message
        BSR.S   WRITES          Write it
        RTS                     Return to main program
* Messages
MESS1   DC.B    'Memory check starting',CR,LF,0
MESS2   DC.B    'Pass completed',CR,LF,0
MESS3   DC.B    'ERROR detected',CR,LF,0
MESS4   DC.B    'Memory check complete',CR,LF,0
        END
```

監訳者注

注1： 「7章のバスエラーとアドレスエラー」参照．

注2： この種のアセンブラはリロケータブル・アセンブラとも呼ばれている．実行可能なオブジェクトを作るには，アセンブラが生成したリロケーション情報を使って，"リンカ"と呼ばれるプログラムによる(リロケーション情報を取り除き，完全なコードとなる)．

注3： つまり，バイト・オフセットでは$104となる．

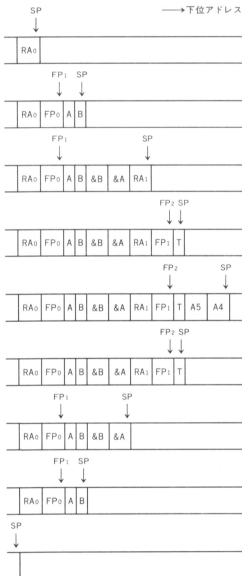
注4： ただ1つのデータ長に対してしか作用しない命令についても，こうした指定を行うのは，冗長であると思える．またある種のアセンブラでは「長さ指定をしてはいけない命令に対する，誤った操作である」として，エラーにしてしまうものがあるので注意する必要がある．

注5： システム・コールの手間が無視できないので高くつくの意味．

注6： 参考までに，具体的なスタック・フレームの使い方をお見せします．これは仮想的なCコンパイラの出力で，手続きMAINが手続きSWAP(2つの変数の値を交換する)を呼び出す様子を示したものです．リスト中で"*"で始まる行はCのソースです．P121，P122にそれぞれ，リスト，図を示します．

```
FP      =        A6
*       main ()
*       {
*                int     a, b;
_MAIN   =        *
~A      =        -2
~B      =        -4
        LINK     FP,#~B              Enter and allocate

*                swap (&a, &b);
        PEA      ~B(FP)              Push address of B
        PEA      ~A(FP)              Push address of A
        BSR      _SWAP               Call SWAP routine
        ADDQ.W   #8,SP               Throw away args.

*       }
        UNLK     FP                  Deallocate and Exit
        RTS

*       swap (x, y)
*       register int     *x, *y;
*       {
*                int     t;
_SWAP   =        *
~T      =        -2
~X      =        A5
~Y      =        A4
        LINK     FP,#~T              Enter and allocate
        MOVEM.L  A4-A5,-(SP)         Save registers
        MOVE.L   8(FP),A5            Load address of X
        MOVE.L   12(FP),A4           Load address of Y

*                t = *x;
        MOVE.W   (A5),~T(FP)         Swap X Y

*                *x = *y;
        MOVE.W   (A4),(A5)

*                *y = t;
        MOVE.W   ~T(FP),(A4)
*       }

        MOVEM.L  (SP)+,A4-A5         Restore registers
        UNLK     FP                  Deallocate and exit
        RTS
```

SP ↓ ——→下位アドレス

RA₀

スタック・フレームを作り，2ワード割付ける

LINK FP, #~B

FP₁ ↓ SP ↓

RA₀	FP₀	A	B

パラメータ(A, Bの番地)をプッシュし，_SWAPを呼び出す

```
PEA  ~B(FP)
PEA  ~A(FP)
BSR  _SWAP
```

FP₁ ↓ SP ↓

RA₀	FP₀	A	B	&B	&A	RA₁

新たにスタック・フレームを作り，1ワード割付ける

LINK FP, #~T

FP₂ SP ↓↓

RA₀	FP₀	A	B	&B	&A	RA₁	FP₁	T

必要なレジスタを退避する

MOVEM.L A4−A5, −(SP)

FP₂ ↓ SP ↓

RA₀	FP₀	A	B	&B	&A	RA₁	FP₁	T	A5	A4

レジスタを復帰する

MOVEM.L (SP)+, A4−A5

FP₂ SP ↓↓

RA₀	FP₀	A	B	&B	&A	RA₁	FP₁	T

スタック・フレームを解放して戻る

```
UNLK FP
RTS
```

FP₁ ↓ SP ↓

RA₀	FP₀	A	B	&B	&A

パラメータを捨てる

ADDQ.W #8, SP

FP₁ ↓ SP ↓

RA₀	FP₀	A	B

```
UNLK FP
RTS
```

SP ↓

スタック・フレームを解放し_MAINを呼び出した手続きに戻る

RAₙ：戻り番地
FPₙ：フレーム・ポインタ
SP：スタック・ポインタ
&A, &B：変数A, Bの番地
A, B, T：変数A, B, Tの実体

CHAPTER 5

算術演算

5.1 加算 124
5.2 減算 126
5.3 値の負数を取る 127
5.4 乗算 128
5.5 レジスタ値の交換 129
5.6 倍長の乗算 130
5.7 除算 135
5.8 倍長の除算 136
5.9 10進演算 139

はじめに

これまでに68000の多くの命令について説明してきましたが，算術演算の方法についてはまだ触れていません．これは偶然こうなったわけではありません．ほとんどのコンピュータでは，算術演算よりもデータの移動や比較の方に，はるかに時間を費やすからです．コンピュータは単に計算機の複雑化したものに過ぎないという考え方は，ひどい時代遅れなものだと言えます．

5.1 加　算

68000ではアドレスの評価において加算を行うので，単純な加算についてはすでに説明したことになります．LEAおよびPEA命令を使って値を加算する方法(ただし，これらの値のうち少なくとも1個がアドレス・レジスタ内にある場合に限る)についてはすでに説明しました．これは便利なトリックではありますが，最も一般的な算術演算は，データ・レジスタ内の値について行われるものです．

加算命令の名前は平凡ですが，ADDといいます．68000の命令セットの多くの命令と同じように，ADD命令のファミリというものが存在します．

基本的なADD命令は，ソースまたはデスティネーションのいずれかに，データ・レジスタを使用しなければなりません．データ・レジスタがデスティネーションである場合は，任意のアドレッシングモードを使用することができます．データ・レジスタがソースである場合は，デスティネーションは．メモリ可変アドレッシングモードを使って指定しなければなりません．ソースがアドレス・レジスタでない場合，バイト，ワード，またはロングのいずれのサイズでも演算を行うことができますが，ソースがアドレス・レジスタである場合，ワードおよびロング・サイズのみ認められます．

演算の結果に応じて，コンディション・コードが影響を受けます．結果が負または0である場合，それぞれNおよびZフラグがセットされ，そうでない場合はクリアされます．オーバーフローが発生した場合には，Vフラグがセットされ，そうでない場合はクリアされます．桁上がりが発生したかどうかに応じ

て，CおよびXフラグの両方がセットまたはクリアされます．

ADD命令は，デスティネーションのデータ・レジスタの下位16ビットまたは8ビットだけを変えるためにも使えます．これと類似した命令であるADDAは，アドレス・レジスタ内の値を加算します．ADDAは，MOVEAと同様に，いずれのコンディション・コードにも影響を与えず，長さはワードまたはロングでのみ使用できます．ワード形式を使用した場合，ソース値のサイズにのみ影響を与え，このソース値は，32ビットに符号拡張され，デスティネーション・アドレス・レジスタに入っている全32ビットに加算されます．最上位ビットがセットされたワード値を加算することにより，負の値がデスティネーションに加算されることになるので，ADDAでワード形式を使う場合には注意が必要です．

ADDAのソースは，もちろん任意のアドレスモードを使って指定することができますが，イミディエイト・データを加算するための，特別な命令形式があります．ADDI命令は，イミディエイト値をソースとして取り，任意のデータ可変アドレッシングモードをデスティネーションとして取ります．つまりADDIは，イミディエイト・データをアドレス・レジスタに加算するためには使用できませんが，デスティネーションが，データ・レジスタである場合は使用できます（むろんADD命令も，この場合に使用できます）．

ADDIは定数値をメモリに加算するために使用します．この命令は3種類のサイズのいずれでも使用でき，ADDと同様の方法でコンディション・コードをセットします．ただし加算される最も一般的な値は1であり，例えばループを1回りするたびに何らかの番地またはレジスタがインクリメントされるといった場合です．命令セットによっては特殊な〝インクリメント〞処理を準備しているものがありますが，68000ではさらに高級なものが用意されています．ADDQ（ADD Quick，素早い加算）命令は，1～8の範囲の数を，任意の可変オペランドに加算します．これは非常に便利な命令であり，例えば，ポインタとして働くレジスタに4を加算する場合がよくあります．

ADDQ命令は，2つの相違点を除いてADDI命令と同じ動作をします．第1の相違点は，ADDQ命令の方が短いので，ADDI命令に優先して使用するべきだということです．特にロング・サイズを使用する場合はそうです，この場合，コンディション・コードはADDI命令とまったく同じ方法でセットされます．第2の相違点は，ADDQがデスティネーションにアドレス・レジスタを使用で

きる点です．この場合，ADDA でイミディエイト・データを使用した場合と同じ動作をします．このときに使用できるサイズはワードまたはロングで（アドレス・レジスタ全体が変えられるので，どちらのサイズを使っても構わない），コンディション・コードは影響を受けません．

一連の ADD 命令の最後の命令は，ADDX（ADD extended，拡張 ADD）です．この命令は，オペランドがデータ・レジスタの対であるか，あるいは，プレデクリメント付きアドレス・レジスタ間接によって指定されるデータの対であるかによって，2つの側面があります．どちらの場合でも，命令はバイト，ワードまたはロングのサイズを使用できます．

ADDX 命令は ADD と同様に 2 つの値を加算しますが，さらに言えることは X フラグの加算も行います．X フラグは通常，ADDX が使用される直前の算術演算によってセットまたはクリアされ，多倍精度算術演算に利用する事ができます．X フラグは算術演算の C フラグと同じ値にセットされますが，MOVE など，C フラグを変化させる他の命令によっては，影響を受けません．

コンディション・コードは，1つの点を例外として，ADD 命令と同じ方法でセットされます．この例外とは Z フラグであり，演算結果が非ゼロの場合，通常の方法でクリアされますが，ただし，結果が 0 の場合はセットされるのではなく変化しません．この機能は通常，多倍精度演算で使用されます．Z フラグは，多倍精度演算を構成する ADDX 命令列の前に，セットされます．もし中間結果のいずれかが 0 以外の場合，このフラグはクリアされ，演算が完了する時点でもクリアされたままです．逆に，演算が完了した時点でも，このフラグがセットされている場合は，すべての中間結果が 0 であり，したがって，その多倍精度演算全体の結果が 0 だということです．

5.2 減 算

ADD 命令にファミリがあるように，SUB 命令にも同様にファミリがあります．基本的な SUB 演算は，ソースまたはデスティネーションとしてデータ・レジスタを使用し，コンディション・コードをセットします．この場合，当然のことながら，桁下がりが発生した場合，C および X フラグがセットされます．

SUBA は，デスティネーションがアドレス・レジスタである場合に使用し，コンディション・コードは変化しません．SUBI はソースがイミディエイト・データである場合に使用し，SUBQ はイミディエイト・データが 1 ～ 8 の範囲内である場合に使用できます．ここでもやはり，SUBQ がワード・サイズで使用され，デスティネーションがアドレス・レジスタである場合，そのワード値は，使用される前に符号拡張されます．

もちろん，SUBX も使用することができます．この命令は，デスティネーションの内容を取り，ソースを減算し，次に X フラグを減算して結果をデスティネーションに入れます．ここでもやはり，オペランドはデータ・レジスタの対，あるいはプレデクリメント付きアドレス・レジスタ間接モードで指定されるデータの対だけを使用することができます．Z コンディション・コードは，結果が 0 以外の場合はクリア，0 の場合は変化しません．

ここではすべての命令について説明していますが，多くのアセンブラは，その状況で使用可能な正しい命令の形式を，自動的に選択します[注1]．アセンブラなしでコードが生成される場合(例：コンパイラで生成される場合)には，適切な選択をすることが重要です．

5.3 値の負数を取る

NEG 命令によって，任意の値の負数をとることができます．この命令は単に，デスティネーションを 0 から減算するだけです．演算のサイズはバイト，ワード，またはロングを使用することができ，デスティネーションは，任意のデータ可変アドレッシングモードを使って指定することができます．

結果が 0 である場合，Z コンディション・コードがセットされ，C および X フラグはクリアされます．結果が 0 以外の場合，Z はクリアされ，C および X はセットされます．N および V は，それぞれ結果が 0 である場合，またはオーバーフローが発生した場合に，セットまたはクリアされます．

NEG 命令の変形は NEGX だけです．この命令は指定された値の負数をとり，その値から X フラグを減算します．コンディション・コード N および V は，NEG と同様の方法でセットされます．Z は結果が 0 の場合はクリアされ，0 以

外の場合は変化せず，ADDX および SUBX と同様です．C および X は，桁下がりが発生したかどうかに応じて，セットまたはクリアされます．この命令は一般的に1ロング・ワード長を越える(つまり5バイト長以上の)多倍精度数値の負数をとる場合に使用します．

5.4 乗　算

68000の加算および減算命令は3種類のオペランド・サイズのいずれに対しても，作用できるという意味で完全と言えます．ところが不都合なことに，乗算および除算に関しては，これが当てはまりません．これら2つの算術演算に認められる命令のサイズはワードだけです．モトローラ社の暫定情報によると，68020では，これらの命令がロング形式で使用可能になるとのことです．

乗算には，MULS と MULU という2つの命令が用意されています．両者の唯一の相違点は，MULS が符号付き算術演算を実行して，符号付きの結果を生成するのに対し，MULU は符号なし算術演算を実行して，符号なしの結果を生成するという点です．

両方の命令とも，任意のデータ・アドレッシングモードをソース・オペランドとして，データ・レジスタをデスティネーションとして取ります．デスティネーション・レジスタの下位16ビットの内容に，ソース・アドレスによって示されるワード値が乗算されます．ソース・アドレスがメモリ番地である場合，その値は，そのメモリ番地で始まる16ビットとなります．ソースがデータ・レジスタである場合，値はそのレジスタの下位16ビットとなります．

結果は，デスティネーション・レジスタ内に，32ビット数として入れられます．N および Z フラグは，通常どおり，結果が0または負の場合にセットされます．符号なしの場合，〝負〟というのは，最上位のビットがセットされている場合を指します．V および C は常にクリアされ，X は影響を受けません．

5.5 レジスタ値の交換

ここで紹介する便利な命令は SWAP です．この命令は単にデータ・レジスタをとり，その上位16ビットを下位16ビットと交換します．この命令は，倍長の乗算および除算ルーチンを提供したり，既に用意されている16ビット演算を使いこなす上で便利なものです．

この命令はコンディション・コードを変化させます．N フラグは，結果の32ビット・データ・レジスタの最上位ビットが1のときセットされ，それ以外の場合はクリアされます．テストされるビットは，演算前の下位ワードの最上位ビットだった点に注意してください．Z フラグは，レジスタ全体が0であるか否かに応じてセットまたはクリアされます．V および C は，常にクリアされますが，X は影響されません．

ここであと2つ，便利な命令を説明しておきます．1つは EXG です．この命令は単に，2個のレジスタに記憶されている値を交換します．2個のレジスタは，両方ともアドレス・レジスタまたはデータ・レジスタでも，またそれぞれ混ぜても構いません．レジスタの内容全体が交換されるので，演算のサイズはロングのみです．コンディション・コードは影響を受けません．

もう1つの命令は EXT です．これはデータ・レジスタの値を符号拡張します．この命令のサイズとしては，ワードまたはロングのみ使用可能です．ワード・サイズを使用した場合，下位バイトの最上位ビットが，レジスタのビット15～8に転送されます．ロング・サイズを指定した場合，下位ワードの最上位ビットが上位ワードにコピーされます．

例えば，符号付き数を表すバイト値が，ワードまたはロング・サイズのオペランドへの加算に先立ってレジスタにロードされる場合，EXT が必要です．ワード・サイズを使って加算を行う場合，レジスタ内のワード値を符号付き数としてセットするために，EXT.W が必要です．同様に，ロング・サイズの加算を行う場合，EXT.L が必要です．EXT.W は，バイト値をワード表現として正しくセットし，EXT.L はワード値をロング表現として正しくセットします．多くの場合において，ワードからロングへの符号拡張は，他の命令の処理中に自動的に実行されます．例えば，アドレス・レジスタに関連する場合がこれにあた

ります．

Xコンディション・コードは，EXT操作によっては影響を受けません．VおよびCは常にクリアされますが，NおよびZは，結果が負であるか，または0であるかにしたがって，セットまたはクリアされます．

5.6 倍長の乗算

倍長の乗算機能が欠落しているため，この仕事を行うサブルーチンを準備する必要があります．このサブルーチンは32ビットデータを取り込み，32ビットの答を出します．2つの大きな32ビット数どうしを乗算した場合，結果は明らかにオーバーフローとなります．最初のプログラム例では，生じるオーバーフローを無視することにします．

乗算を実行するために，まず倍長の乗算を筆算で行う方法について考えてみましょう．計算機をふだん使っていない人は，2桁の10進数を乗算する場合のアルゴリズムをよく知っていると思います．ほとんどの人は2桁の10進数の乗算なら，頭の中でできますが，それ以上複雑になった場合は，紙と鉛筆を使わざるを得ません．2つの数ABとCD(個々の英字は，単一の10進数の1桁を表す)を乗算する場合を考えます．これは次のように計算します．

```
                    A      B
            ×       C      D
       ----------------------
                  D×A    D×B
  C×A             C×B
  ---------------------------
  C×A     D×A+C×B        D×B
```

1桁の数字どうしならいつでも乗算することができるので，この乗算をより単純な乗算と加算に分けていくことができます．最上位桁は，上位桁どうしを乗算した結果であり，最下位桁は下位どうしを乗算した結果です．結果の中の残りの桁は，交差する項どうしの乗算結果の和です．単純な乗算の結果が2桁

の数になる場合，オーバーフローの処理を忘れずに行わなければなりません．これは余分に生じた桁を次の桁に繰り上げることによって行います．

これとまったく同じアルゴリズムが，倍長の乗算ルーチンでも必要とされます．68000 は，2 つの 16 ビット数の乗算をいつでも実行できるので，倍長の乗算を，実行可能な乗算といくつかの加算に分割します．個々の 32 ビット数を，2 つの 16 ビット数から構成されているものと見なし，その数が 32 ビット・レジスタ R に入っている場合，その 2 つを RH および RL で表します．ここでは，この方式を使用し，前の図の C×A で表される 3 番目の桁がオーバーフローする場合のことは無視します．したがって，2 つの数が初めにレジスタ D1 と D2 に入っている場合，結果は次のように表されます．

```
RL  =  D1L * D2L
RH  =  (D2H * D1L) + (D1H * D2L) + carry from RL
```

これを 68000 のコードにしてみましょう．レジスタ D1 および D2 に入っている 2 つの数を乗算し，結果を D1 に入れるルーチンを作成します．

```
MUL      MOVEM.L D2-D4,-(SP)  Save registers
         MOVE.W  D1,D3        D1L into D3
         MOVE.W  D2,D4        D2L into D4
         SWAP    D1           D1H into D1
         SWAP    D2           D2H into D2
* Create the products
         MULU    D3,D2        D2 = D1L*D2H
         MULU    D4,D1        D1 = D2L*D1H
         MULU    D4,D3        D3 = D2L*D1L
* Add cross terms ignoring overflow
         ADD.W   D1,D2        D1 = D2L*D1H+D1L*D2H
* Place cross term into high digit of result
         SWAP    D1           D1H = cross term
         CLR.W   D1           Clear D1L
* Insert bottom digit
         ADD.L   D3,D1        D1L and D1H now correct
         MOVEM.L (SP)+,D2-D4  Restore registers used
         RTS
```

ここで，MULU 命令は 2 個の 16 ビット値を取り，32 ビットの結果を出すことを思い出してください．1行目は使用されるレジスタを保存し，次の 4 行では入力された数を 4 個の 16 ビット単位に分割します．MULU では各レジスタの上位ワードは無視されるので，何が入っていても問題にはなりません．MOVE.W 命令は下位ワードに対してのみ影響を与え，次に SWAP 命令を使って上位ワードを，レジスタの下位部分に入れます．

次の 3 行は，16 ビットの入力値の積として，3 個の 32 ビット値を結果として出します．次に ADD.W 命令を使って，**交差積**(cross product)の加算を行います．これでオーバーフローが発生するかもしれませんが，それは無視します．この 16 ビットの和は，結果レジスタ D1 の上位ワードに移動し，下位ワードはクリアされます．

必要とされる最後の動作は，結果の下位ワードを結果レジスタの正しい位置に挿入することです．この処理は下位桁から上位桁への桁上がりを必要とするので，D3 から D1 へ MOVE.W によっては実行できません．D1L と D2L の積は D3 の上位ワードに入れられます．D3 と D1 の単純な ADD.L によって，下位ワードが正しく結果に挿入されることが保証されます．この命令によって発生するオーバーフローは，やはり無視されます．

次のルーチンは，D1 および D2 に入っている 2 個の 32 ビット数を取り，レジスタ対 D6，D7 に，64 ビットの結果を与えます．言い換えると，D6 には結果の上位部分が入り，D7 には下位部分が入ります．結果を得るために前のアルゴリズムを，次のように拡張しなければなりません．

```
D7L  =  D1L * D2L
D7H  =  (D2H * D1L) + (D1H * D2L) + carry from D7L
D6L  =  D1H * D2H + carry from D7H
D6H  =  carry from D6L
```

このために算術命令の実行中に桁上がりが発生したかどうかを示す，X フラグを利用します．

```
LMUL     MOVEM.L D3/D4,-(SP)  Save registers
         MOVE.L  D1,D3
         MOVE.L  D2,D4
         SWAP    D3           D3 = D1H
         SWAP    D4           D4 = D2H
* Create products
         MOVE.W  D1,D7        D7 = D1L
         MULU    D2,D7        D7 = D2L*D1L
         MOVE.W  D3,D6        D6 = D1H
         MULU    D4,D6        D6 = D2H*D1H
         MULU    D1,D4        D4 = D1L*D2H
         MULU    D2,D3        D3 = D2L*D1H
* Add cross products together
         ADD.L   D3,D4        X bit relevant now
* Handle low order part of cross product
         MOVE.W  D4,D3        D3L = D4L
         SWAP    D3           D3H = D4L
         CLR.W   D3           Clear D3L
* Handle high order part of cross product
         CLR.W   D4           Clear D4L
         ADDX.W  D3,D4        Set D4L to state of X bit
         SWAP    D4           D4 = high 17 bits of cross
*                             product
* Add low order cross terms in D3H to D7
         ADD.L   D3,D7        X bit relevant
* Add high order of cross terms in D4
         ADDX.L  D4,D6        Include carry from
*                             previous ADD
* D6,D7 now holds unsigned 64-bit product
         TST.L   D1           Check D1 negative
         BPL.S   LMUL1        Branch if not
         SUB.L   D2,D6        Subtract D2 from answer
LMUL1    TST.L   D2           Check D2 negative
         BPL.S   LMUL2        Branch if not
         SUB.L   D1,D6        Subtract D1 from answer
LMUL2    MOVEM.L (SP)+,D3/D4  Restore registers
         RTS
```

このルーチンの最初の数行は，作業レジスタ D3 および D4 を保存し，次に，オペランドの上位ワードを作業レジスタの下位ワードに入れます.

次の各行は積項を作成します.結果の最下位桁は D7 に入るため即時に計算されます．同様に最上位桁は D6 に入ります．作業レジスタ D3 および D4 を使って，2 個の交差積を入れます．これらは加算されて D4 に 32 ビットとして入れられ，X フラグによって桁上がりが発生したかどうかが示されます．後の段階で D6 に X フラグを加算するようにしなければなりません.

次の各行では交差積を取り扱います．交差積の下位ワードは，結果の 2 番目

の桁位置，すなわちD7Hに入れ，また，D7Hに入っている下位積からの繰上がり桁も含めなければなりません．したがって，下位ワードをクリアしてD4LをD3Lに移動し，レジスタの上下半分を交換し，下位ワードをクリアして，D3Hの中の交差積の下位ワードを出します．この値はすぐにD7に加算されますが，それによってXフラグ(交差積の項が加算されたときに桁上がりが発生したかどうかをまだ示している)が影響を受けるので，この処理はまだ実行することはできません．

交差積の上位部分は，D6に加算されるに先立って，作業レジスタの下半分に入れられます．ここでもXフラグに注意しなければなりません．D4の下半分は，**CLR.W**でクリアされ，XフラグはD4Lに入れられます．この処理は，**ADDX**でイミディエイト値0を引数として使用したいところですが，**ADDX**は，2つのデータ・レジスタまたは2つのアドレス・レジスタを指定し，かつプレデクリメントモードでのみ使用可能なのでこの使い方はできません．そのため，ここでは前の2つの命令によってD3Lが0にクリアされているので，**ADDX.W**をD3およびD4に対して使用しています．この処理が完了すると，D4Hには，交差積の上位桁が入り，D4Lには，Xフラグの状態に応じて，0または1が入ります．ここで単純な**SWAP**を使用することにより，D4に交差積の上位17ビットが確実に入ります．

これでほとんど完成です．D3をD7に加算することにより，結果の下位2桁が正しいことが保証されます．また，D7からD6へ，桁上がりがある場合は，Xフラグがセットされます．D4のD6への**ADDX.L**により，この桁上がりビットと，交差積の上位部分が加算されて，前に生成された上位積と加えられます．こうしてD6とD7に64ビットの結果が得られます．

このプログラムは，正の数については正しく動作しますが，実際には，入力値に対して，符号なし乗算を実行したのです．つまり，例えば−1と2を乗算しようとした場合，実際には$FFFFFFFFを$2で，符号なし算術演算を使って乗算したことになります．この結果，D6には$1，D7には$FFFFFFFEが入ります．この値は，符号付き算術演算を行った場合に予想される値，−2とは異なっています．

2の補数演算を行っているという事実に基づけば，これを訂正するのは簡単です．Mを2^{32}とするとき，−AはM−Aで表されるので，このとき次のような

関係が成立します.

$$(-A)\times B=(M-A)\times B=(M\times B)-(A\times B)$$

負の数Aと正の数Bの符号なしの積は,符号付きの積にM×Bを足したものと同じになります.M×Bの乗算を実行するのは非常に単純であり,Bの64ビット表現を32桁分左へシフトする処理が伴うだけです.Bは単一の32ビット・レジスタに入っており,結果はレジスタの対D6,D7に入っているので,結果の上位レジスタ,すなわちD6からBの値を減算します.元のオペランドはまだD1とD2に入っているので,D1がもともと負であった場合は,D6からD2を減算しなければなりません.D2が負であった場合は,D6からD1を減算します.この訂正によって符号付き算術演算ルーチンが完了します.

5.7 除 算

前にも述べたとおり,68000には,倍長形式の乗算と除算がありません.符号付きおよび符号なし除算を取り扱う,2つの除算命令が準備されています.これらは両方とも,ソースとしてワード・サイズの値を取り,任意のデータ・アドレッシングモードを使って指定することができます.デスティネーションは,ロング・サイズのデータ・レジスタでなければなりません.このデータ・レジスタに入っている32ビット値全体が,ソースとして指定されたワード値によって除算されます.DIVU命令は符号なし算術演算を使ってこの除算を行い,DIVS命令は符号付き算術演算を使用します.

どちらの場合も,2つの結果が生成されます.デスティネーションの下位ワードは,商にセットされます(1ワードに収まるものと仮定して).上位ワードは,整数の剰余にセットされます.DIVSの場合,この剰余は分子(すなわちデスティネーション)と同じ符号になります.

NおよびZステータス・フラグは,商が負または0であるかに応じて,通常どおりセットまたはクリアされます.Cフラグはクリアされ,Xフラグは変化しません.商が16ビット値よりも大きい場合は,オーバーフローが検出され,

Vフラグがセットされます．ただし，68000が除算命令の処理中に，オーバーフローの検出が生ずる場合がありますが，この場合，結果およびN,Zフラグの状態は不定となります[注2]．TRAPV命令(第7章で説明)を使って，オーバーフローが実際に生じた場合にトラップを起こすことができます．またソースが0である場合は，〝0による除算〟トラップが発生します．

5.8 倍長の除算

DIVSおよびDIVUは，結果が16ビットより小さい場合にしか作用しないので，倍長の除算ルーチンが必要になります．これは，前述の倍長の乗算ルーチンよりもやや難しくなります．次のルーチンはArthur Norman博士によるもので，D1に入っている32ビットの分子を，D2に入っている32ビットの分母で除算します．32ビットの商がD1に戻され，D2には整数の剰余が入ります．

最初のセクションでは符号を取り扱い，ルーチンの本体では符号なし算術演算で処理が行えるようにします．まず，分母が負かどうかをチェックします．負である場合，分母を正にし，除算を実行した後，結果の符号を反転します．

```
DIV       TST.L   D2          Check denominator < 0
          BPL.S   DIV00       No
          NEG.L   D2          make denominator positive
          BSR.S   DIV00       Do division as if positive
          NEG.L   D1          Now negate the answer
          RTS                 And return
```

この次のケースでは，分子が負で分母が正である場合を取り扱います．この場合，分子を正にし，除算を実行した後，商と剰余の両方を反転します．

```
DIV00     TST.L   D1          Check numerator < 0
          BPL.S   DIVU        Both operands positive
          NEG.L   D1          Make numerator positive
          BSR.S   DIVU        Perform division
          NEG.L   D1          Correct sign of quotient
          NEG.L   D2          Correct sign of remainder
          RTS                 Complete
```

次のセクションはD1とD2が0以上，$80000000以下である場合の符号なし数の除算を取り扱います．D2が実際に0である場合は〝0による除算〟トラップが発生します．手間を省くために，多くの簡単なケースについてチェックを行います．最初のチェックは分母が16ビット未満のケースであり，標準的なDIVU命令を使用することができます．この場合，標準サブルーチンDIVX(除算を実行し，剰余を正しくセットする)にジャンプします．

```
DIVU    CMPI.L  #$FFFF,D2    Test if D2H is zero
        BLS.S   DIVX         D2 < 16 bits,
*                            use subroutine
```

この段階で，さらに2つの特殊なケースについてチェックします．分子が分母より小さい場合，結果は0になり，分子と分母が等しい場合は，商は1になります．

```
        CMP.L   D1,D2        Check if D2 <= D1
        BEQ.S   DIV01        D1 = D2, simple case
        BLS.S   DIV02        Difficult case
* Here D1 < D2, so the result is zero
        MOVE.L  D1,D2        Get remainder correct
        MOVEQ   #0,D1        Zero result
        RTS
* Here D1 = D2, so the result is 1
DIV01   MOVEQ   #0,D2        Zero remainder
        MOVEQ   #1,D1        Result is 1
        RTS
```

より一般的なケースとして分母が16ビットより大きい場合があります．分子が32ビットに収まる場合，結果の商は16ビットのオブジェクトになります．分子と分母の双方をある**重み付け因子**(scaled factor)で除算することによって必要な商への近似を行います．この重み付け因子は重み付けされた16ビットに収まるように選びます．こうして重み付けしたオペランドに対して，標準的な除算を行うことができますが，このとき選択する重み付け因子はそれ自体が16ビットに収まりきるものでなければならず，適切な精度で近似を行えるものでなければなりません．実際には1＋(D2/$1000)を重み付け因子として使用していますが，この場合許し得るD2の最大の値が$80000000であり，そのため生じる最大の重み付け因子は$8001となるので常に16ビットに収まります．

```
DIV02   MOVEM.L D3-D5,-(SP)   Save work registers
        MOVE.L  D2,D3         Save denominator
        CLR.W   D3            Clear D3L
        SWAP    D3            D3 = D2 / $10000
        ADDQ.L  #1,D3         d3 = 1 + (D2/$10000)
* Scale factor in D7. Scale numerator and denominator
        MOVE.L  D1,D4         D4 = numerator
        MOVE.L  D2,D5         D5 = denominator
        MOVE.L  D3,D2         Scalefactor into D2 for
*                             DIVX
        BSR.S   DIVX          D1 = D1 / Scalefactor
        MOVE.L  D5,D2         Replace denominator
        DIVU    D3,D2         D2L = D2 / Scalefactor
* D2 should now fit into 16 bits
        DIVU    D2,D1         Divide scaled terms
```

この時点で，D1Lには求めるべき商の第1近似が入ります．この商の近似値に，もとの分母を掛け，もとの分子と比較することによって，結果をチェックします．同時に，剰余も生成することができます．商が正しくない場合，1を加算または減算して，正しい結果になるまで再試行します．

```
        ANDI.L  #$FFFF,D1     D1H = 0
DIV03   MOVE.L  D5,D2         Restore original
*                             denominator
        MOVE.L  D5,D3         Into D3 as well
        SWAP    D3            D3L = D2H
        MULU    D1,D2         D2 = D1*D2L
        MULU    D1,D3         D3 = D1*D2H, D3H is zero
        SWAP    D3            Move into high digit
        ADD.L   D3,D2         Get product, no carry
*                             possible
        SUB.L   D4,D2         Subtract original
*                             numerator
        BHI.S   DIV04         Overshot, remainder
*                             negative
        NEG.L   D2            Change sign
        CMP.L   D2,D5         Compare with original
*                             denominator
        BHI.S   DIV05         OK, remainder is in range
        ADDQ.L  #1,D1         Increment quotient
        BRA.S   DIV03         Try again
DIV04   SUBQ.L  #1,D1         Decrement quotient
        BRA.S   DIV03         Try again
* Got it!
DIV05   MOVEM.L (SP)+,D3-D5   Restore registers
        RTS
```

これで残った処理はサブルーチン DIVX を詳細化するだけになりました．このサブルーチンは，もとの商が 16 ビットに収まる場合に使用し，また，より難しい場合に分子を重み付けする目的でも呼び出します．もとの D1 を D2 で除算した値にし，D2 を整数の剰余にします．D1 と D3 の下位ワードを保存するために MOVEM.W を使用している点に注意してください．レジスタを復帰するために MOVEM.W を使用しない理由は，レジスタを 1 個ずつ取り上げた方が都合が良いということもありますが，主にレジスタの復帰のために MOVEM.W を使用すると，レジスタの全内容が変化するという理由によります[注3]．

```
DIVX      MOVEM.W D1/D3,-(SP)  Save D1L AND D3L
          CLR.W   D1           Clear D1L
          SWAP    D1           D1 = D1H
          DIVU    D2,D1        D1L = D1H/D2
          MOVE.W  D1,D3        Save partial result
          MOVE.W  (SP)+,D1     Retrieve D1L
* D1H holds D1H rem D2, D1L as on entry
          DIVU    D2,D1        D1L =(D1L+(D1H rem D2))/D2
          SWAP    D1           D1L now holds remainder
          MOVEQ   #0,D2        Clear D2
          MOVE.W  D1,D2        Remainder into D2
          MOVE.W  D3,D1        D1L = high order quotient
          SWAP    D1           Swap to get 32bit quotient
          MOVE.W  (SP)+,D3     Restore D3L
          RTS                  All done
```

5.9 10進演算

これまでの説明は，2 進数値——2 の補数による 2 進形式の数——の算術演算に関するものでした．この種のデータに対する算術演算は高速ですが，人間が読む 10 進数値と，コンピュータが読む 2 進数値の間で変換を行うのは，どちらかというと面倒です．

いくつかの高級言語(例：COBOL)では，10 進算術演算または 2 進算術演算のどちらで演算を行うかをプログラマが，選択できる機能が用意されています．2 進数を使用することの利点は，計算が速いことですが，10 進数から 2 進数への変換プロセスは時間がかかります．10 進算術演算を使用することの利点は，10 進数入力から 10 進形式でコンピュータに値を読み込むのが非常に速く，しか

も簡単だということですが，逆に不利な点は計算が遅いということです．

68000では10進算術演算用に3つの命令が用意されています．**ABCD**(Add Binary Coded Decimal，2進化10進数の加算)，**NBCD**(Negate Binary Coded Decimal，2進化10進数の負数をとる)，そして**SBCD**(Subtract Binary Coded Decimal，2進化10進数の減算)です．これらの演算はバイト・オペランドをとります．このオペランドは，2進化10進数(**BCD**)形式で2桁の10進数を表現します．個々の〝**ニブル**(nibble)〟(4ビット)は，0～9の範囲の10進数を入れるために使用されます．したがって，10進数16は，2進数形式では$10として，**BCD**では$16として記憶されます．

一般的に外部媒体からの数は10進形式で一度に1文字ずつ読み込まれます．10進数全体を2進数に変換するためには，次のようなルーチンを使用しなければなりません．

```
RDN     MOVEQ   #0,D1       Clear total
        MOVE    D1,D0       Clear all of D0
RDN1    BSR     RDCH        Get character in D0
        SUB.B   #'0',D0     Subtract character 0
        BMI.S   RDN2        Negative - no more digits
        CMP.B   #9,D0       Check valid digit
        BGT.S   RDN2        No more digits
        MULS    #10,D1      Multiply old total by 10
        ADD.L   D0,D1       Add this digit to total
        BRA.S   RDN1        Get next digit
RDN2    RTS                 Return with total in D1
```

このルーチンは，何らかの外部媒体から読み込まれた10進数を2進形式で組み立てます．一度に1文字を得るために，サブルーチン**RDCH**を使用します．文字は，有効な数字であるかどうかがチェックされ，対応する2進数に変換されます．以前の合計値に10を乗算し，その合計に新しい数字を加算します．このルーチンは，1語に収まりきる数だけを読み取ります．これより大きな数が必要な場合は，倍長の乗算サブルーチンを呼び出すよう修正しなければなりません．数を再び10進数に変換して書き出せるようにするには，多少複雑なプロセスを行う必要があります．

この処理には時間がかかります．そして，この方法で得られた数に対して行うべき動作は，これらの数に他の値を加算することである場合が多々あります．一例として，数字の長い列を読み取っていき，それらの総計を求める場合を考

えてみます．そのためには10進数を読み込んで，それを2進数に変換するサブルーチンを使って，この処理を行うルーチンを書くことができます．

```
COUNT   MOVEQ    #0,D4        Grand total in D4
        MOVE.W   #4,D5        5 numbers to read
CNT1    BSR.S    RDN          Get number in D1
        ADD.L    D1,D4        Add to total
        DBRA     D5,CNT1      Get next number
```

この場合，2進演算を用いるよりはむしろ10進演算を使用した方が良いでしょう．処理に使用される2つの数を入れるため，作業領域を使います．次のルーチンは，1つの数を読み込み，BCD形式の結果をA1によって指される8バイトの領域に入れ，右詰めされた16桁の欄から，数を読み取るものとします．そのため，読み込まれた数は，用意されたBCD領域にぴったり入ります．そうでない場合は，前に数を読み込んだ領域をクリアする必要があります．先行空白桁は0として取り扱われ，欄に入っているのが数字と空白だけかどうかはチェックしません．ルーチンからのリターン時にはA1は作業領域の直後の番地を指します．

```
DRDN    MOVEM.L D0/D1,-(SP)  Save registers
        MOVEQ    #7,D1        Initialise counter
DRDN1   BSR      RDCH         Get character
        ASL.B    #4,D0        Move up to high nibble
        MOVE.B   D0,(A1)      Place in memory
        BSR      RDCH         Get next character
        AND.B    #$F,D0       Mask to low nibble
        OR.B     D0,(A1)+     Install in memory
        DBRA     D0,DRDN1     Loop until all done
        MOVEM.L (SP)+,D0/D1  Restore registers
        RTS                   Return
```

最初の数行では，レジスタを保存し，カウンタを初期設定します．RDCHを呼び出し，リターン時には，16進数または空白の文字表現がD0に入っています．ASCIIコードでは，数字は16進数$30～$39，空白は$20で表されます．欄に空白または数字だけが入っている場合は，下位4ビットを見るだけで，その数に対する正しい値を得ることができます．欄に空白または数字だけが入っているものと仮定して，文字表現を左側へ4ビットだけシフトし，下位ニブルをクリアして，上位ニブルを10進数値にセットします．この値が，現在A1が指

している番地に記憶されます．

続けて RDCH を呼びだすと次の数字または空白が返されます．このとき，この値と$F の AND をとり，上位ニブルをクリアします．次に結果とメモリとの OR をとります．これは下位ニブルを組み立て，必要な記憶形式をとるためです．ポストインクリメント・アドレッシングを使用しているので，この段階で A1 は次のバイトを指しています．処理の終了時には，A1 はメモリ内の 8 バイト高い位置を指した状態になっており，記憶領域には，その数の BCD 表現が入っています．この数は，32 ビット 2 進形式で通常記憶できる数より，大きい場合もある点に注意してください[注4]．

次のルーチンは，ABCD を使って 10 進数を加算します．この方法は前出の例で，ADD を使って 2 進数を加算した場合と同じ方法です．

```
SIZE     EQU      8              Number of bytes to hold
*                                each number
COUNT    LINK     A0,#-SIZE*2    Allocate room for two
*                                numbers
         LEA.L    -SIZE(A0),A1   Set A1 to top of second
*                                area
* Set first area to BCD zero
CNT1     CLR.B    -(A1)          Decrement A1 and zero byte
         CMPA.L   A7,A1          Check if end of first area
*                                reached
         BNE.S    CNT1           Continue until all cleared
         MOVE.W   #4,D5          Set up counter
* This loop performs the totalling
CNT2     LEA.L    -SIZE(A0),A1   Set A1 to base of second
*                                area
         MOVEA.L  A1,A2          Set A2 to top of first
*                                area
         BSR.S    DRDN           Read number
* A1 and A2 now point to the end of decimal values
         SUB.B    D0,D0          Clear X bit
CNT3     ABCD     -(A1),-(A2)    Add two bytes
         CMPA.L   A7,A2          Finished yet?
         BNE.S    CNT3           Continue
         DBRA     D5,CNT2        Loop until all done
```

このプログラムの最初の部分は，スタックから必要とする作業領域を割り付けます．これは 2 個の数については充分であり，この例では，総計は 8 バイトに収まりきるものと想定します．総計を入れるために最初の領域を使い，読み

込んだ数を入れるために2番目の領域を使います.

最初の小型のループは, A7が作業領域の下限を示していることに注意して, 総計用の全バイトを0にセットします. 総計を行う主ループの最初の2つの命令は, A1を入力値領域の先頭にA2を結果領域の直後に単純にリセットします. DRDNによって数を読み取ると, A1は入力値領域の直後を指すようになります.

次のループは10進算術演算を使って入力された数を結果に加算します. ABCDはソースとデスティネーションを加算しますが, 同時にXフラグも含めて行うので, この場合Xステータス・フラグの状態が, 非常に重要な意味を持ちます. Xフラグは10進キャリーが発生した場合にセットされ, 繰り返し実行される加算が正しく動作します. ループを開始する前に, Xフラグをクリアしなければなりません. そのため, D0自身からD0のバイト内容を減算し, これが確実にクリアされるようにします. Xフラグをこれよりもっとエレガントにクリアする方法については後で説明します[注5].

ループを1回まわるたびに, 2個の10進数値からの各1バイトがXフラグとともに加算されます. 10進キャリーが発生した場合, CおよびXフラグがこのことを次回に反映させます. Zフラグは, いずれかのバイトが0以外の値である場合にクリアされますが, 結果が0である場合は影響を受けません. SUB命令を使ってZフラグをセットしているので, 加算ループ全体の完了時には, 全バイトが0になった場合にのみ, Zフラグがセットされます. この場合, もし希望するなら, 結果が0の場合にのみ何らかの動作を行うために, Zフラグをテストすることができます. 他の2つのコンディション・コード(NとV)は, ABCD命令の実行後は不定となります. ループが完了したかどうかを調べるCMPA命令によっては, Xフラグは影響を受けません.

読者の皆さんは, ABCDの双方のオペランドに対してプレデクリメント付きアドレス・レジスタ間接モードで使用していることに気付くでしょう. これは, ABCD命令で許されるただ2つのアドレスモードの1つであり, もう1つは双方ともデータ・レジスタをとります. このメモリを使う形式(つまり前者)が一般的に最も便利であり, プレデクリメントモードとなっているのはバイトを加算する順序が下位桁から上位桁に向かって行われなければならないためです. データ・レジスタの場合は, 個々のバイトを2個のレジスタに入れることが許されます. どちらの場合でも命令のサイズはバイトです.

SBCD命令もこれと非常に似ています．やはり同じ2つのアドレスモードしか使用できず，デスティネーションバイトには，デスティネーションのもとの10進数値から，ソースの10進数値とXフラグを引いた値が入ります．コンディション・コードも同様の方式で変えられますが，ただしCおよびXは，10進桁下がりが発生した場合にセットされます．

最後の10進命令がNBCDであり，10進数値の負数をとります．実際に，NBCDはNEGXと似ており，デスティネーション・バイトの負数がとられ，次にXフラグがそれから減算されます．コンディション・コードは，SBCDと同じ方式でセットされます．前出の2つの命令と異なり，NBCDのオペランドとしては，任意のデータ可変アドレスモードを使用することができます．

前に説明したCMPM命令は，10進演算ではありませんが，10進演算を取り扱う場合に便利です．この命令はポストインクリメント付きアドレス・レジスタ間接モード[注6]でしか使用できませんが，メモリに入っている2個の10進数を比較して，それらが等しいかどうか調べる場合に便利です．

10進演算処理における最終的な目的は，BCD形式で記憶されている数を書き出すことです．このようなルーチンは，前出のDRDNサブルーチンの逆になります．数はA1を使ってBCD記憶領域の開始番地のポインタとして渡され，WRCHを使って出力します．A1は，BCD領域の直後を指した状態で終わります．この単純な例では，先行する0をすべて出力していますが，より洗練された版では，先行する0を空白に変換することもできます．

```
DWRN    MOVEM.L D0-D2,-(SP)  Save registers
        MOVEQ   #7,D1        Initialise counter
        MOVEQ   #12,D2       Set up shift value
DWRN1   MOVE.L  #$30300,D0   Set up D0
        MOVE.B  (A1)+,D0     Extract two decimal
*                            characters
        ROR.W   #4,D0        Move nibbles around
        BSR     WRCH         Print first digit
        LSR.L   D2,D0        Shift down other character
        BSR     WRCH         Print second digit
        DBRA    D1,DWRN1     Loop
        MOVEM.L (SP)+,D0-D2  Restore registers
        RTS                  And return
```

このルーチンの主ループは，まずD0を$30300という値にセットしていますが，その理由はあとで明らかになります．次に，出力するべき2桁の10進数を

含む BCD 領域から1バイトを取り出し，これを D0 の下位バイトに入れます．したがって，そのバイトに$56 が入っていた場合，D0 にはこの段階で$30356 が入ることになります．

ROR.W 命令は下位ワードを4ビットだけ回転し，最下位のニブルを下位ワードの上半分に入れ，このワードの残りの部分を右側へ4だけシフトします．この例では D0 にはこの段階で$36035 が入ります．下位バイトには$35 が入りますが，これには数字5の ASCII 表現であり，**WRCH** への呼び出しによって書き出されます．

LSR.L 命令は，D0 の全内容を，右側へ12だけシフトするために用います．シフト量は，8以下の場合に限り，イミディエイト値で表現できるので，D2(既に12に初期設定している)を使用しなければなりません．これにより，この例では$36 という値が下位バイトに入り，これは〝6〟の ASCII 表現に相当します．次にこれが書き出され，もし必要ならループバックして全桁を出力します．

文字形式と **BCD** との間で変換を行う処理は，2進数との間における同様の変換より高速ですが，多くのコンピュータでは，〝パック〟および〝アンパック〟と呼ばれる命令が準備されています．これらは，1つの形式から他の形式へ，1つの簡単な命令によって変換するものです．実際に，68000 の初期のドキュメントでは，これらの命令についての説明がありましたが，実装はされませんでした．ここで 68020 では **PACK** および **UNPK** が使用できるようになるとお約束いたしましょう．

監訳者注

注1：　このようなアセンブラでは，ソースおよびデスティネーションの組合せから適切なものを選択する．例えば，

```
ADD     Dn,An     ──→  ADDA   Dn,An
ADD     #val,Dn   ──→  ADDI   #val,Dn
                       または
                       ADDQ   #val,Dn
                       （1≦val≦8のとき）
```

注2：　ただしオペランドは変化しない．

注3：　MOVEM.W でレジスタをロードするときは，符号拡張でロング値となることは，既に述べたとおり．

注4：　例えば，BCD 形式 16 桁，9999999999999999 を考えてみると，2 進形式 32 ビットによる最大値は，4294967295（符号付きならば，2147483647）にすぎない．

注5：　「6章のはじめに」参照．

注6：　比較を上位桁から行うのが賢いやり方，それゆえポストインクリメントモードとなるのが合理的である．

CHAPTER 6 論理演算

6.1 シフトとローテイト 151
6.2 16進表現への変換 154
6.3 単一ビットの演算 155
6.4 フリーエリア割付けパッケージ 157

はじめに

ここまでの章で，数字または文字がレジスタおよびメモリ中のビット・パターンでどのように表され，それをどうやって適切に処理するかをみてきました．信頼性を高めるには3つ以上の状態を持つものよりも，ただ2つの状態(例：ON と OFF)を持つ電子素子を使うのが簡単です．それゆえ，コンピュータでは2進表現が使われています．数字や文字が記憶される厳密な方法については，普通は知る必要がありません．しかし，2進表現を利用し，その値が単なるビットの集合であることを理解した上で操作するためには，やはり知っておくと便利です．このような取扱い方法のことを，算術演算と区別して**論理演算**といいます．単一のビットは，1または0ではなく，論理値の**真**または**偽**を持つものと考えられます．

おそらく，最も簡単な論理演算は，オペランドの全ビットを反転する処理でしょう．

```
NOT.L   D3
```

これはD3の中の1のビットをすべて0に，0のビットを1に変換します．他の論理演算と同様，NOT はデータ可変オペランドに対してのみ，使用することができます．これは，アドレス・レジスタにはアドレスだけが入るという規約を補強するものであり，また，アドレスに対して論理演算を実行すると，プログラムが不正確で見通しの悪いものになりがちだという点もあります．しかし，ときには，アドレスに対してビット処理を実行することが望ましい場合もあります．これについては本章の後半の記憶域割付けルーチンの中で例を示します．

NOT の他には2個のオペランドをとる論理演算が，複数用意されています．これらの論理演算は，バイトまたはワード全体を単一の値として操作するのではなく，各オペランドの対応するビットが1つに集められ，それをまとめて操作するという点で算術演算とは異なっています．この種の論理演算は，8,16または32ビットの1ビット演算を並行して行うのと同じです．

OR 演算では，ソースまたはデスティネーションのうち，いずれか一方に1が

セットされていれば，結果のビットが1にセットされます．D1の下位バイトが2進数11001100であり，D2の下位バイトが11100001である場合，

```
OR.B     D1,D2
```

D2の下位バイトに，11101101が入ります．OR の考え方を別の言い方で表すと，1に対して OR をとるとビットは常にセットされ，0に対して OR をとるとビットは変化しません．したがって，レジスタの上位半分を残りの部分に影響を与えずに1にセットするには，次のように指定します．

```
OR.L     #$FFFF0000,D3
```

OR と相補的な論理演算が AND です．AND 演算の結果は，第1オペランドおよび第2オペランドの両方のビット位置に1がある場合にビット値1となります．D1 および D2 の値が前の例と同じである場合，

```
AND.B    D1,D2
```

D2の最下位バイトに11000000が入ります．0に対して AND をとると0になり，1に対して AND をとるとビットが変化しないという点で，AND は OR の逆であると言えます．AND 命令は値の一部を**マスキング**する(すなわち，その他のビットに影響を与えることなく不要のビットを0にセットする)場合に便利です．一例として，何かの計算の結果，D4の下位バイトに8ビットの値が入り，他のビットに関しては，その中の値について何も保証できない場合を考えてみます．D4全体の値が必要とされる結果だけになるように，これら不要のビットをクリアするには，次のように指定します．

```
AND.L    #$000000FF,D4
```

2組のビットを合せるための3番目の命令が**排他的** OR(EOR)命令です．この命令は，2つのオペランド・ビットが異なっている場合に結果のビットを1にセ

ットします．逆に2つのビットが両方とも等しい場合に，結果は0となります．D1に11001100，D2に11100001が入っている場合，

```
EOR.B    D1,D2
```

結果としてD2に00101101が入ります．

別の見方をすれば，0に対してEORをとるとビットは変化せず，1に対してEORをとるとビットが反転します．したがって，全ビット1に対するEORは，NOTと同じ意味です．

EORに関して認められるオペランドの形式は，ソースがデータ・レジスタでなければならないためANDおよびORの場合とはやや異なります[注1]．ANDおよびORでは，オペランドの少なくとも一方がデータ・レジスタでなければなりませんが，それはソースでもデスティネーションでも構いません．ソースが一定のビット・パターンである場合に使用するために，これら3種類の命令には，イミディエイト形式(ANDI，ORI，およびEORI)があります(ただし，イミディエイトのソース・オペランドは，普通のANDおよびOR形式でも使用できます)．これらの命令の特徴は，デスティネーションにステータス・レジスタ(SR)を使用できるという点です．演算のサイズがバイトである場合，ステータス・レジスタの下位バイトだけが影響を受けます．つまり，コンディション・コード・レジスタ(CCR)です．サイズがワードである場合，ステータス・レジスタ全体が使用され，演算が特権化されます[注2]．3種類の命令のこれらの形式を使用することにより，特定のステータスおよびコンディション・コード・フラグをセット(ORI)，クリア(ANDI)，または反転(EORI)することができ，それ以外のビットに影響を与えません．例えばキャリーフラグをクリアするには，次のように指定します．

```
ANDI.B   #$FE,CCR
```

また，トレースモードをセットするには次のように指定します．

```
ORI.W    #$8000,SR
```

6.1 シフトとローテイト

これまでに，個々のビットをそのビット位置で操作する方法について説明しました．次に，レジスタまたはメモリ内でビット・パターンを移動する方法を説明します．この処理を行う命令には4タイプあり，それぞれの命令に左への移動と右への移動の形式があります．シフト命令はすべて，3つの形式のオペランドを取ります．オペランドがメモリ内にある場合，演算サイズは常にワードであり，シフトは1ビット分です．オペランドがデータ・レジスタに入っている場合，3種類全部の演算サイズが認められ，シフトは一定量(1～8ビット)または，他のデータ・レジスタによって与えられる数になります．

論理シフト命令LSLは，オペランドの全ビットを左側へ移動し，右側に0を埋めます．A1によってアドレスされるメモリ・ワードに1011111111111111が入っている場合，次のように指定すると，

```
LSL.W   (A1)
```

このワードは0111111111111110にセットされます．桁上がり(C)および拡張(X)フラグは失われたビットでセットされ，NおよびZフラグは，結果の値から通常の方法でセットされます．

前にも述べたとおり，データ・レジスタの中の値は，一度に1ビット以上ずつシフトすることができ，この数は2通りの方法で指定することができます．1～8ビットの範囲の一定量のシフトは，次のように，イミディエイト・データで表します．

```
LSL.L   #4,D2
```

もう1つの方法は，シフト・カウントを他のデータ・レジスタで与える方法です．

```
LSL.L   D1,D2
```

使用されるカウントは，レジスタを64で割った剰余です．

LSR 命令は右側への論理シフトを行い，左側に 0 を埋め，C および X フラグには最後に右端から掃き出されたビットが入ります．

2 進数を 1 ビット左へシフトした場合，その値を 2 で乗算したことになります．逆に，1 ビット右へシフトした場合は，その値を 2 で除算して剰余を捨てたことになります．したがって，値を 2 の累乗(2, 4, 8, 16, …)で乗算または除算する方法としてシフトを使用することができます．この方法で乗算を行うために LSL 命令を使用することができますが，ただし LSR 命令による除算では，負の数については正しい結果は得られません．すなわち，0 が左側へシフト・インされ，符号ビットをクリアするからです．

この問題を回避するため，あと 2 つのシフト命令が準備されています．これらは，算術シフト ASL および ASR で，オペランドは 2 の補数形式の数とします．これらの命令は，LSL および LSR と，オペランド形式はまったく同じですが，ただし符号ビットとコンディション・コードの取扱いが違っています．左シフトの場合はこの違いは些細なものです．LSL は，V(オーバーフロー)フラグを常にクリアするのに対し，ASL は，算術オーバーフローが発生した場合にこのフラグをセットし，それ以外の場合にクリアします．メモリ・オペランド，またはレジスタ内の最終的なビット・パターンは，両方の命令とも同じです．

符号ビットの取扱いの違いは右シフトに対してのみ影響を与えます．左側にシフト・インされたビットは，もとの符号ビットのコピーであるため，ASR は正の数，負の数ともに作用できることが保証されます．正の数の場合，LSR と ASR の作用は同じですが，負の数の場合，ASR は，左側に 0 だけではなく 1 をシフト・インします．例えば，D1 の最下位バイトが11101100(＝10進数の－20)である場合，次のように指定すると，

```
ASR.B    #2,D1
```

このバイトは，11111011(＝10進数の－5)にセットされます．結果の小数部分はもちろん失われ，除算の結果は負の無限値に向かって切り捨てられます．

● 5を1ビット右へシフト　＝5／2＝2
● －5を1ビット右へシフト＝(－5)／2＝－3

算術および論理シフトでは両方ともシフト・アウトされるビットを失います．代替の方法は，ローテイト演算(巡回シフトともいう)であり，これを使用した場合，オペランドの端から出てしまったビットが，もう一方の端から再び現れます．したがって，情報が失われることはなく，シフトを十分な回数実行することによって(1バイトにつき8回)，オペランドをもとの状態に復元することができます．

ローテイト命令には2種類あります．両方とも，他のシフト命令と同じ形式をとり，右，左の変形があります．

ROLおよびRORは指定された量だけ，オペランド値をローテイト(回転)し，Cフラグには，オペランドの端から他方の端へ回された，最後のビットのコピーが入ります．NおよびZフラグは，結果値からセットされ，Vは常にクリアされ，そしてXは影響を受けません．

ROXLおよびROXRもこれと非常によく似ていますが，シフト・アウトした各ビットがXフラグに入り，もとのXフラグの値がもう一方の端に回ります．これらの命令は，単純なローテイトよりも，オペランドを復元するのに，1ステップよけいにかかります(1バイトにつき9ステップ)．

ROXLおよびROXR命令の重要性は，オペランドに入れられるビットの値が，前の命令によって決定される唯一のシフト演算であるという点です．そのため，32ビットより大きいオブジェクトのシフトにも使用できます．例えば，64ビットのデータがD1およびD2にある場合，この値全体に対して左論理シフトを行うには，次のように指定します．

```
        LSL.L    #1,D2          LS half: lost bit goes
*                                        into X
        ROXL.L   #1,D1          MS half: get bit from LS
*                                        half from X
```

6.2 16進表現への変換

これまでに説明してきた論理演算を使って，レジスタに入っている数を文字による16進表現に変換するコードを書くことができます．

個々の16進数1桁は数の4ビットに対応し，このことをニブル(1バイトの半分)と言う場合もあります．ローテイトを使って，各ニブルを順にレジスタの最下端に移し，次に **AND** 演算を使ってマスクします．

この操作により，0～15の数が得られます．そしてこの数を利用して16エントリのテーブルの中から適切な文字を選択することができます．

```
* The number to be converted to characters is in D1
* A0 points to an 8-byte buffer for the character form

        MOVEQ    #7,D0            Use D0 as loop count

LOOP    ROL.L    #4,D1            Get next nibble to bottom
*                                 of D1
        MOVE.B   D1,D2            Copy two lowest nibbles
        ANDI.L   #$F,D2           Mask low nibble
        MOVEA.L  D2,A1            Need it in addr register
        MOVE.B   CHARTAB(A1),(A0)+ Put corresponding
*                                 character in next buffer
*                                 position and step
        DBRA     D0,LOOP          On to next nibble

* Exit here: conversion complete

*       : :
*       : :

CHARTAB DC.B    '0123456789ABCDEF' Conversion table
```

6.3 単一ビットの演算

データ・レジスタまたはメモリのバイト内の単一のビットに対して作用する命令には，5種類あります．そのうちの1つ，TASは，やや特殊なので後述します．その他の4つ，BTST，BCLR，BSET，およびBCHGは，同じ形式のオペランドを取るファミリーを形成します．これらの4つの命令はそれぞれ，単一のビットに対して作用し，そのビットの位置は，メモリ・バイト中のそのビットの番号，またはレジスタ・オペランドによって指定されます．ビットは，オペランドの最下位(最右端)から順に0,1,2,…と番号付けされます．したがって，1つのレジスタには，ビット番号0～31があり，1バイトには，ビット番号0～7があります．

BTST命令は，単に指定されたビットをテストし，Zフラグをその値にセットします．これ以外のコンディション・コード・フラグは影響を受けません．ビット番号はイミディエイト値，またはデータ・レジスタのどちらかで指定します．デスティネーション・オペランドは，イミディエイト・アドレッシングモードを除く，任意のデータ・アドレッシングモードを使用することができます．オペランドがデータ・レジスタである場合，ビット番号は，32の剰余によって指定されます．すなわち，指定した数を32で除算したときの剰余が使用される数となります．したがって，次の2つの命令は，

```
BTST    #3,D7        and
BTST    #35,D7
```

両方とも，D7のビット番号3をテストします．オペランドがメモリである場合，ビット番号は8の剰余がとられます．

このファミリーの他の3つの命令も，指定されたビットをテストしますが，そのビットを変化させる場合もあるので，デスティネーション・オペランドはデータ可変アドレッシングモードでなければなりません．BCLRはビットを0にクリアし，BSETはビットを1にセットし，BCHGはビットの元の値を反転させます．

メモリはバイト単位で構成されているので，ビット演算ではバイト全体を読

み取り，そのバイトを変化させ，次に全ビットを書き戻さなければなりません．すなわち，BCLR，BCHG，または BSET を使って，メモリ空間に割付けられた周辺装置の制御レジスタ中のビットを変化させた場合，予測不能な結果が生じる場合があります．というのは，番地を読み取る動作自体によって，周辺装置内で何らかの動作を引き起こす原因となり得るからです[注3]．この場合は，まず必要とされるビット・パターンを組み立て，次に MOVE を使ってこのような制御レジスタをセットした方が安全です．

TAS(テスト・アンド・セット)命令も，オペランド中の1ビットだけに影響を与えます．この命令は，オペランド・サイズが常に〝バイト〟であり，テストされ，セットされるのは，常にそのバイト中のビット7なので，他のビット命令より融通性が低いと言えます．N および Z フラグは両方とも，オペランド・バイトの元の値に応じてセットされます．TAS は，任意のデータ可変オペランドを指定することができますが，ただし TAS の重要性は，メモリ内のバイトにアクセスする方式にあります．TAS は，いわゆる〝リード・モディファイ・ライト〟メモリ・サイクルを使用します．すなわち，TAS は，その命令を実行中，メモリの制御権を保持し続けるので，TAS がオペランド・バイトを検査しセットしている間は，他の命令がそのオペランド・バイトを見たり，変化させることはできません．第1章で述べたように，複数のコンピュータが同じメモリを共有する場合，これは強力な操作です．というのは，この命令により，コンピュータがメモリ内のフラグバイトを使って，それらのコンピュータ間で共有している資源が現在使用されているかどうかを示すことができるからです．このようなフラグのことを，**セマフォ**(semaphore)といいます．各コンピュータは，他のコンピュータによってセマフォ操作を妨げられることなく，セマフォをセットしたり，またその状態を調べ得る能力を持つことが重要です．

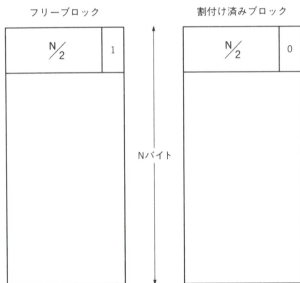
コンピュータに多かれ少なかれ，独立的に走るプログラムがいくつか入っている場合，単一のコンピュータ中でもセマフォが必要となる場合があります．多くのオペレーティング・システムでは，複数のプログラムが同時に活動状態(active)になることを認め，それらのプログラムの間で，プロセッサの時間を分け合うようにしています．単純なシステムでも，割込みルーチンのコード(第7章を参照)は，メイン・プログラムの実行中にランダムな時点で実行することができます．単一のコンピュータでは，BCLR，BCHG，および BSET(TAS も同

様)は，すべて単一の命令で〝テスト・アンド・セット〟型の動作を行うので，セマフォ操作に使用することができます．すなわち，あるプログラムがセマフォを調べ，セットしている間は，他のプログラムがセマフォを変化させることはできないからです．

6.4 フリーエリア割付けパッケージ

単純なプログラムでは，一般的に各種のデータ構造やバッファなどのために，これからどの程度のデータ記憶域が必要になるかを前もって予測し，それに応じて使用可能なメモリを分割することができます．しかし，多くのプログラムでは全記憶域を異なる使用目的で分割する必要があったり，またある種のデータ構造のために必要とされる量は，プログラムが実際に実行されるまで分からないことがあったりします．このような場合，記憶域を確保しておき，必要がなくなったときに解放するという機構があれば便利です．このような機構はフリーエリア割付けパッケージを形成するルーチンの集まりとして提供されます．

単純なパッケージは，メモリの割付け用と解放用の2つのルーチンしか含んでいません．記憶域を編成する1つの方法を以下に説明します．使用可能な記憶域がすべて1つの連続する領域にあると仮定し，領域全体を各ブロックに分割します．各ブロックにはヘッダワードがあり，第一にブロックの長さを示し，第二にそのブロックがフリーか使用中かを示します．一番最後のブロックは，0の入ったヘッダだけで構成されています．

ここで次のような規則を作ります．ブロックの長さはすべて4の倍数となるバイト長とし，ロングワードのヘッダを持つものとします．こうするとシステム・メモリと同じ大きさまでのブロックが許され，ブロックとブロックの間に無駄な隙間を残すことはありません．さらに，全ブロックが偶数アドレスで開始するようにして，ヘッダをロング・サイズの命令で操作できるようにすることができます．ブロック長はすべて4の倍数なので，その下位2ビットは常に0です．したがって，最下位ビットをブロックがフリーかどうかを示すフラグとして使用することができます．

各ブロック(ヘッダも含めて)のブロック長はNバイト

記憶域を割り付けるルーチンを GETBLK と呼ぶことにします．このルーチンは〝ファースト・フィット〞アルゴリズムを使ってメモリを割り付けます．

すなわち，各ブロックを見ていき，その中から充分な大きさを持つ最初のフリーブロックから割付けを行います．ブロックを見ていくなかで，隣接し合うフリーブロックがあれば，それを結合します．多数のブロックが割り付けられ，再びそれが解放された場合，いくつかのフリーブロックが互いに隣り合っている可能性があります．

後の段階での GETBLK 要求に対し，充分な大きさを持つ単一のブロックがないという理由だけで拒絶するのは馬鹿々々しいことです．しかし，このような要求があるまで，隣接しあうブロックを結合する作業をする必要はありません．

```
*            GETBLK
*
*  Routine to allocate an area of store
*
*  Entry:  D1 = number of bytes required
*
*  Exit:   A0 = address of first byte of allocated block,
*               or zero if allocation failed
*          D0 = error code: 0 = block allocated
*                           1 = insufficient free store
*                           2 = block list found to be
*                               corrupt

GETBLK  MOVEM.L A1/D1-D3,-(SP) Save work registers
        ADDQ.L  #3,D1          Add 3 to number of bytes
*                              wanted
        ANDI.B  #$FC,D1        Round to multiple of 4
        ADDQ.L  #4,D1          Block size = (rounded)size + 4
        BLE     GBC7           Error if negative

GBCRTY  LEA.L   BLKLIST,A0     Get start of store chain

* Search down the chain for a free block and
* amalgamate any adjacent free areas.

GBC1    MOVE.L  (A0),D2        D2 = size + marker of block
        BLE.S   GBC6           End of list (or error)
        BCLR.L  #0,D2          Test the marker and clear it
        BNE.S   GBC2           Jump if the block is free
        ADDA.L  D2,A0          A0 = address of next block
        BRA.S   GBC1           Continue down list

* Have found a free block

GBC2    MOVE.L  A0,D3          D3 = address of free block

GBC3    ADDA.L  D2,A0          A0 = address of next block
        MOVE.L  (A0),D2        Get size and marker of next
*                              block
        BMI.S   ERRSTORE       Jump if loop in free store
        BCLR.L  #0,D2          Test the marker and clear it
        BNE.S   GBC3           Jump if block free - carry on
*                              if allocated (or end of chain
*                              reached)
* Now D1 = size required in bytes
*     D3 = address of start of free area
*     A0 = address of end of area
* Amalgamate the group of free blocks

GBC4    MOVE.L  A0,D2          Copy end address
        SUB.L   D3,D2          D2 = amalgamated size in bytes
        BSET    #0,D2          Set free marker
        MOVEA.L D3,A1          Get start in address register
        MOVE.L  D2,(A1)        Amalgamate free blocks
        BCLR    #0,D2          Unset free marker for
*                              arithmetic
```

```
        SUB.L   D1,D2           Split block
*                               (D2 = size of excess)
        BLT.S   GBC1            Can't be done
        BEQ.S   GBC5            Exact fit

* Must make new block for upper part

        SUBA.L  D2,A0           A0 = address of upper part
        BSET    #0,D2           D2 = size of upper part + mrkr
        MOVE.L  D2,(A0)         Plant in upper block

GBC5    MOVE.L  D1,(A1)         Plant the size
*                               (marker bit zero)
        ADDQ.L  #4,D3           D3 = addr of allocated space
        MOVEA.L D3,A0           Put in result register
        CLR.L   D0              No errors
        BRA.S   GBEXIT          Return

* Error or end of store chain reached

GBC6    BMI.S   ERRSTORE        Loop in store chain

* Not enough free store for request

GBC7    MOVEQ   #1,D0           Insufficient store code
GBERREX SUBA.L  A0,A0           Clear result register
GBEXIT  MOVEM.L (SP)+,A1/D1-D3 Restore work registers
        RTS                     Return

* Error exit

ERRSTORE MOVEQ  #2,D0           Corrupt store chain code
        BRA.S   GBERREX         Take error exit
```

割付け済みの記憶域のブロックを解放するルーチンを FREEBLK と呼ぶことにします．このルーチンの仕事は非常に単純です．ヘッダワードの最下位ビットをセットして，ブロックがフリーであることを示すようにするだけです．しかし，解放するアドレスが，確かに GETBLK によって以前割り付けられたものであることを確認するため，いくつかの単純なチェックを行います．渡されたアドレスが偶数であるかチェックし(ブロックがすべて偶数アドレスであるという規則を作ったので)，ヘッダワード内のフラグビットを検査して，ブロックが割付け済みであることを確認します．

最もよく発生するプログラミング・エラーは，ブロックを2回フリーにしてしまうことですが，この単純なテストによってそれは検出されます．さらに FREEBLK は，アドレスまたはヘッダの最上位バイト内のいずれのビットもセットされていないことをチェックします[注4]．ブロックのリスト全体を走査して，アドレスが

本当に割付け済みのブロックを参照していることを確認するチェックなどのより精巧な(そして時間のかかる)チェックは，この場合省略しています．

FREEBLK は，引数として 0 を与えられた場合，即座に戻ります．このことによって，たとえ GETBLK が記憶域の割付けに失敗したとしても[注5]，GETBLK の結果が常に FREEBLK に対する有効な引数であることが保証されます．

```
*            FREEBLK
*
*  Free store allocated by GETBLK
*
*  Entry:  D1 = block address
*
*  Exit:   D0 = 0  Block freed
*             = 1  Does not appear to be an allocated block

FREEBLK MOVEM.L D1/D2/A0,-(SP) Save work registers
        CLR.L   D0             Set 'ok' result
        TST.L   D1             Look at block address
        BEQ.S   FBEXIT         Return if zero

        SUBQ.L  #4,D1          Point at block header word
        MOVE.L  D1,D2          Copy the header address

* Inspect address given: it should be even, and should
* have no bits set in the top byte.

        ANDI.L  #$FF000001,D2 Check address if of form
*                              0000 0000 dddd ... ddd0
        BNE.S   FBERR          It isn't

        MOVEA.L D1,A0          Get into address register
        MOVE.L  (A0),D1        Look at first word of block
        ANDI.L  #$FF000001,D1 Check header is of form
*                              0000 0000 dddd ... ddd0
        BNE.S   FBERR          It isn't

        BSET    #0,3(A0)       Set 'free' marker bit
FBEXIT  MOVEM.L (SP)+,D1/D2/A0 Restore work registers
        RTS

FBERR   MOVEQ   #1,D0          Error result
        BRA.S   FBEXIT         Return
```

最初のフリー記憶域の初期設定は次のように行います．この記憶域はひとかたまりの大きなフリーブロックから構成され，ワード0で終端されています．

```
BLKLIST DC.L    BLKEND-BLKLIST+1 Size + 'free' marker
        DS.L    1000             Some free store
BLKEND  DC.L    0                End marker
```

監訳者注

注1：　EORのビット・パターンをよく見てみると，CMP命令と同じグループ(上位4ビットが$B)に入っている．言わば，CMP命令の隙間を利用したために，EOR命令に許されるアドレッシングモードが制限されたというべきでしょう．

注2：　もし特権化しないと，ユーザーモードで走っているプログラムからステータス・レジスタのシステムバイトを自由に操作することができてしまい，システムが安全なものではなくなってしまう．

注3：　読み出し動作によって，ポート・ステータス・レジスタの状態を変化させるといったI/Oポートが，よくみられる．ステータス・レジスタの読み出しによって割込み要求フラグをクリアする(つまり，その動作が割込みサービスを行った対応であるとして)といった動作が典型的な例である．

注4：　つまり，アドレスが24ビットに収まり切るかどうかを検査する．

注5：　最後のブロックは，ヘッダだけのダミー・ブロックであり，ヘッダの内容は0であったことを思い出してほしい．

CHAPTER 7 例外処理

7.1 例外処理ベクタ 165
7.2 ユーザーモードとスーパーバイザモード 167
7.3 例外処理の動作 168
7.4 例外処理ルーチン 169
7.5 割込み 172
7.6 外部リセット 173
7.7 不正命令と未実装命令 173
7.8 トラップの原因となる命令 174
7.9 特権違反 176
7.10 トレース 178
7.11 バスエラーとアドレスエラー 179
7.12 例外処理の順位付け 181
7.13 メモリサイズ判定ルーチン 182

はじめに

これまでに見てきたプログラム例ではすべて，次に実行する命令のアドレスは，現在実行中の命令によって(暗黙的または明示的に)決定されています．一般的に，一連の命令列の次の命令がこれに当たりますが，ただし，分岐，ジャンプ，またはリターン命令によって，他の位置へ実行を移すことができます．

本章では，これ以外の方法によって制御が移される場合について説明します．このような状況を〝**例外処理**(exceptions)〞といいます．例外処理を使用する目的には，2つあります．1つは，何らかの事象(ユーザーがターミナルのキーを押した場合など)が発生した場合に即座に動作が行えるようにするためです．もう1つの目的は，エラーの検出と，適切なエラー処理手続を起動する機能をコンピュータに備えるためです．一例として，不正命令が検出された場合が挙げられます．

例外処理が行われる場合，68000はプログラム・カウンタとステータス・レジスタの現在値を保存し，下位のメモリ・アドレスにある**例外処理ベクタ**で与えられるアドレスから，実行を続けます．保存された情報は，あとの時点で，割込みがかかった場所から実行を再開するために使用されます．この作用は，2つの命令の間で1つのサブルーチンを呼び出す場合と類似しています．

例外処理要求が生じる場合としては2通りあります．**内部的**にプロセッサ自体が異常な状況を検出した場合と，**外部的**に他の何らかのハードウェアがプロセッサによる処理を必要としている場合です．内部的なものを**トラップ**というのに対し，外部的なものによる例外処理を一般的に**割込み**といいます．

▶外部的な例外処理

- 割込み
- バスエラー
- 外部リセット

▶内部的な例外処理

- 不正命令
- 未実装命令(unimplemented instruction)

・アドレスエラー
・ユーザーモード状態での特権化命令の使用
・トレース
・0による除算
・TRAP，TRAPV，CHK

7.1 例外処理ベクタ

例外処理要求が生じると，プロセッサはその例外処理を行うためにユーザーによって準備されたルーチンを呼び出します．一般的に，ユーザーはプロセッサに対し，そのルーチンのアドレスを見つける方法を与えなければなりません．ある種のコンピュータでは，どの例外処理要求に対しても同一のルーチンが呼び出され，そのルーチンのアドレスは，固定的なメモリ番地に入っています．このルーチンの最初の部分では，何らかのシステム・レジスタを調べて，実際に何が発生したのかを見つけなければなりません．

68000では，これより一般性を持たせた処理方法として，例外処理のそれぞれの型，および各外部装置用に独立したルーチンを使用することができます．これらのルーチンすべてのアドレスを入れるために，メモリの最下位1024バイトが確保されています．各アドレスは，**例外処理ベクタ**と呼ばれる4バイトのスロットに入っています．各ベクタには番号が付いており，その番号は，そのベクタのバイト・アドレスを4で除算したものです．ベクタの位置および番号を次の表で示します．

例外処理ベクタ番号は，すべての内部割込み，および自動ベクタ機構を使用する外部割込み[注1]に対しては暗黙的です．これ以外の割込みを発生させる回路は，68000に対しベクタ番号を発生させなければなりません．このベクタ番号の選択はシステムの設計者が行います．

68000からの3本のファンクション・コード出力ラインを使って，メモリを個別のアドレス空間に分割している場合，リセット・ベクタを除くすべてのベクタが，スーパーバイザデータ空間から取られます．リセット・ベクタは，スーパーバイザ・プログラム・アドレス空間から取られます．

ベクタ番号	アドレス(16進)	例外処理または割込みの型
0	0	リセット：初期SSP
1	4	リセット：初期PC
2	8	バスエラー
3	C	アドレスエラー
4	10	不正命令
5	14	0による除算
6	18	CHK命令
7	1C	TRAPV命令
8	20	特権違反
9	24	トレース例外処理
10	28	未実装命令（1010）
11	2C	未実装命令（1111）
12 : 14	30 : 38	不定．モトローラが将来的な機能拡張のために確保
15	3C	未初期化割込み
16 : 23	40 : 5C	不定．モトローラが将来的な機能拡張のために確保
24	60	疑似割込み
25	64	レベル1割込み自動ベクタ
26	68	レベル2割込み自動ベクタ
27	6C	レベル3割込み自動ベクタ
28	70	レベル4割込み自動ベクタ
29	74	レベル5割込み自動ベクタ
30	78	レベル6割込み自動ベクタ
31	7C	レベル7割込み自動ベクタ
32 : 47	80 : BC	TRAP ＃0命令 TRAP ＃15命令
48 : 63	C0 : FC	不定．モトローラが将来的な機能拡張のために確保
64 : 255	100 : 3FF	ユーザー割込みベクタ

7.2 ユーザーモードとスーパーバイザモード

第1章で述べたように，68000はユーザーモードまたはスーパーバイザモードのいずれかで命令を実行することができます．これら2つのモードは，異なる特権レベルに対応しています．スーパーバイザモードのほうが特権レベルが高く，オペレーティング・システムの一部を形成しているプログラムは通常，このモードで走らなけらばなりません．このモードでは，任意の命令を実行することができます．

他のすべてのプログラムは，ユーザーモードで走らせます．このモードでは，いくつかの重要な命令は禁止されており，それらの命令を実行しようとすると，トラップが発生し，オペレーティング・システムに制御が戻されます．禁止されている命令は，2つのカテゴリーに分けられます．コンピュータの動作に干渉する命令(例：STOP および RESET)と，プログラムをスーパーバイザモードにし，かつまた自分自身の命令実行を継続する命令です．

プロセッサ・チップには，現在のモードを示す1本の出力ライン[注2]が有ります．この出力ラインを使って，2通りの方法でスーパーバイザに属するメモリを保護することができます．1つは，ユーザーモード時に特定領域へのアクセスを禁止する目的で，ハードウェア内のメモリ・アクセスのたびにこのラインをチェックする方法です．このラインを使用するもう1つの方法は，スーパーバイザで使用可能なメモリと，ユーザー・プログラムで使用可能なメモリを完全に分離するものです．そのため，ユーザーモードにおける1000番地は，スーパーバイザモードにおける1000番地とは異なるメモリ番地をアクセスすることになります．この方法によれば，スーパーバイザ専用のメモリは，ユーザー・プログラムからは見ることができません．

アドレス・レジスタ7は，ある種の命令と，例外処理および割込み中に暗黙的にスタック・ポインタとして使用される点で，特別なレジスタです．またこのレジスタは，名前が2つの物理的レジスタ，すなわちユーザー・スタック・ポインタ(USP)とスーパーバイザ・スタック・ポインタ(SSP)に対応すると言う点で特殊です．A7に対する参照によってアクセスされるものは，プロセッサの状態によって異なります．SP という名前は，現在のスタック・ポインタを示

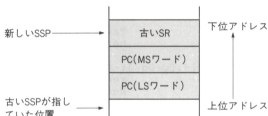
すために使用される場合がしばしばあります．したがってSSPは，ユーザーモードではアクセス不可能です．

しかし，オペレーティング・システムがUSPを読み込みセットする必要があるので，USPにアクセスするための特別な命令があります．この命令は，USPをソースまたはデスティネーションのいずれかに指定した，MOVE命令の特殊なケースです．この命令は特権化されていますが，それは保護の理由によるものではなく，この命令をユーザーモードで使用するプログラムは，何か馬鹿馬鹿しいことをしているからです[注3]．

ユーザーモードからスーパーバイザモードへ変化できる唯一の方法は，トラップまたは割込みが発生した場合です．すなわち，状態の変化は常に例外ベクタ中のトラップ・ベクタによって決定されるアドレスへのジャンプを伴います．この記憶域をユーザーモードによるアクセスから保護することにより，システム・プログラムへのジャンプを伴わずにスーパーバイザモードへ入ることが不可能であることを保証し，さらに，ユーザープログラムができる処理を制限することができます．

スーパーバイザモードからユーザーモードへ戻る方法にはいくつかあります．スーパーバイザは，プロセッサ・ステータス・レジスタ(SR)を直接更新する命令を実行することができるので，モードを制御するビットを単にユーザーモードにセットすることができます．スーパーバイザはさらに，RTE命令(後述)による例外処理からのリターン動作の一部として，ユーザーモードをセットすることができます．

7.3 例外処理の動作

プロセッサがとる動作は，あらゆる種類の例外処理を通じて類似しています．すべての場合[注4]において，プログラム・カウンタとステータス・レジスタの現在の値が保存されます．したがって割込みをかけられたプログラムは，結果的に何もなかったように実行を再開することができます．これらの値は，次の図のように，システム・スタックの3ワードに保存されます．

例外処理が発生したときに，プロセッサがどちらのモードにあったかに関わりなく，使用されるのは常に**スーパーバイザ・スタック**(USPではなく，SSPによって指される)である点に注意してください．アドレスエラーとバスエラー例外処理は，これよりも多くの情報を保存します(後の説明を参照)．多くの場合において，保存されたプログラム・カウンタは，例外処理が発生しないときの次の命令を指しています．68010と68020ではスタックにもう一語余分に積むので注意してください(これは例外処理ベクタ・オフセット)．これにより，同じコードを共有して異なる例外状況を処理するのが簡単になります．

ステータス・レジスタは，例外処理が生じたときの値が保存されたあと，標準状態にセットされます．スーパーバイザモード・ビットは，例外処理ルーチンが常にスーパーバイザモードで開始するよう，1にセットされます．トレース・ビットは0になるため，メイン・プログラムがトレースされていても(後の説明を参照)，通常どおり例外処理を行うことができます．3ビットの割込みマスクは，リセット例外処理と割込みによってのみ影響を受けます．リセットの場合，このマスクは7にセットされ，割込みの場合は，その割込みの優先順位レベル(後の説明を参照)にセットされます．

7.4 例外処理ルーチン

例外処理を行うプログラムは，それぞれの例外処理が発生した場合に呼び出されるルーチンのアドレスを例外処理ベクタに挿入する処理から始まります．例外処理ルーチンの概要は次のとおりです．

```
ENTRYPT MOVEM.L Dp-Dq/Ar-As,-(SP) Save regs used below
*         :
*         :                       Take action necessary to
*         :                       handle exeption
*         :
        MOVEM.L (SP)+,Dp-Dq/Ar-As Restore all saved
*                                 registers
        RTE                       Return to interrupted
*                                 program
```

例外処理は，不定の時点で，命令と命令の間で発生する可能性があります．正しく実行再開するためには，もとのステータス・レジスタと，すべてのアドレスおよびデータ・レジスタの内容を保存しておくことが重要です．ステータス・レジスタは自動的に保存されますが，他のレジスタが破壊されないことを保証するのは，例外処理ルーチンの仕事です．このための最も簡単な方法は，**MOVEM** を使って，ルーチンで使用されるレジスタをスタックに保存し，最後にそのスタックを使って，すべてのレジスタを回復することです．

例外処理手続きからのリターン動作の残りの部分を行うのが **RTE** 命令です．**RTE** 命令は，プログラム・カウンタおよびステータス・レジスタが，上記の順序でスタックに入っているものと見なします．そして，それらをポップし，割込みをかけられたプログラムの実行を再開します．**RTE** による動作は，次のように指定した場合と似ています．

```
MOVE.W  (SP)+,SR
RTS
```

ただし，**RTE** 命令を，これら2つの命令で置き換えることはできません．というのは，**RTE** は常に**スーパーバイザ・スタック**からプログラム・カウンタをポップするのに対し，**RTS** は，現在のプロセッサモードに応じて，ユーザーまたはスーパーバイザ・スタックからプログラム・カウンタをポップします．**SR** への **MOVE.W** 命令によって，モードがスーパーバイザからユーザーに変化した場合，**RTS** は，誤ったスタックに対して作用することになります．

RTE は，ステータス・レジスタを直接変更可能にし，そのため制御をはずれてスーパーバイザモードに入る方法が与えられることになるので，ユーザーモードでの **RTE** の使用は禁止されています．

RTE と非常に類似している非特権化命令が **RTR** です．**RTR** もやはり，スタ

ック上の値をポップして，プログラム・カウンタおよびステータス・レジスタをセットします．唯一の相違点は，RTRの場合，ステータス・レジスタのユーザーバイト(コンディション・コードを含んだ半分)のみを，セットするという点です．RTRは，ステータス・レジスタ用のスタックのワード全体を取りますが，実際に使用するのは下位バイトだけです．したがってRTRは，次の手続きに似ています．

```
        MOVE.W  (SP)+,CCR
        RTS
```

RTRの使用例の1つとして，呼び出し時のコンディション・コードを保存するサブルーチンでRTSの代わりとして使用する場合があります．このようなサブルーチンの形式は次のとおりです．

```
SUBNAME MOVE.W  CCR,-(SP)     Save cond codes
*         :
*         :
*         :
        RTR                   Restore condition codes
*                             and return
```

RTE，RTR，およびRTSのもう1つの使用法は，ジャンプ命令としてです．プログラムの作成時にはわからない，つまり実行時にならないとわからないアドレスに対するJMPを行うには，そのアドレスを，アドレス・レジスタ(An)に入れ，JMP(An)とするものです．別のやり方として，アドレス・レジスタを使わずにどこかへジャンプしたい場合は，すべてのレジスタを特定の値にセットした後，スタックに飛び先の番地を入れ，RTSを使用します．

```
        MOVE.L  destaddr,-(SP) Store destination addr
*         :
*         :                   Set all registers
*         :
        RTS                   Jump
```

また，RTSを使う代わりに，ジャンプの前にコンディション・コード，または(スーパーバイザモードで)ステータス・レジスタ全体をセットして，RTRお

よび RTE を使ってこのようにすることもできます．

7.5 割込み

割込みは，外部装置がプロセッサによる動作を要求するための手段です．装置は，1～7の範囲の割込み優先順位レベルをプロセッサに提示することにより，割込みを要求します．現在のプロセッサの優先順位レベルより高い場合，あるいは要求されたレベルが7である場合に，割込みが受け入れられます．このようにレベル7は，他のコンピュータでいうところの，**ノンマスカラブル・インタラプト**(NMI)として作用します．割込みを要求する回路は，それがベクタ番号付き割込みか，それとも**自動ベクタ**が使用されるのかどうかも示さなければなりません．後者の場合，要求された優先順位レベルに応じて，プロセッサは，25～31の範囲のベクタ番号を暗黙的に使用することになります．多数の装置が同じ優先順位レベルで割込みを行う場合は，一般的にベクタ番号付き割込みを使用します．

割込み処理は，例外処理の通常の手続きに続いて行われます．この場合，まずプログラム・カウンタとステータス・レジスタがスタックに保存され，ステータス・レジスタは標準状態にセットされます．保存されたプログラム・カウンタは，割込みが発生しなかったら，次に実行されたはずの命令を指しています．ステータス・レジスタ中の優先順位(割込みマスク)は，受け入れた割込みのレベルにセットされます．この目的は，同じか，より低いレベルの割込みが発生するのを防止し，より高い優先順位の割込みを認めるためです．コンピュータの周辺装置によっては，割込み要求に対して素早く応答しないと，データを消失してしまうものもありますが，逆に緊急性をあまり必要としないもの，あるいはまったく必要としないものもあります．コンピュータ・システムの設計者が割込み優先順位を選択する場合，緊急性を要するものに対する割込みの処理を優先させるよう選択することができます．割込みルーチンのプログラミングを単純化するために，現在のレベルと同じか，より低いレベルの割込みは禁止されています．もし禁止されていないとしたら，割込みサービス・ルーチンに割込みがかけられ，同じルーチンが再び呼び出されるという事態もあり得

ます．このような場合，手続きが任意の資源(メモリまたはポート)の排他的な使用権(exclusive access)[注5]を持っているとして設計されているとしたら，混乱が発生します．

7.6 外部リセット

リセット例外処理は，プロセッサ外部の回路によって発生します．これは，プロセッサを初期化して動作を開始するため，あるいは，他の方法では回復できない壊滅的状況(crash)のあとで再開するために使用します．リセットの時点で使用されていたすべての状態は，消失し初期化されます．

リセットが他の例外処理と異なっている点は2つあります．第一に，スタック・ポインタが有効なアドレスを参照しないので，スタックには何も保存されません．第二に，例外ベクタは8バイト長であり(4バイトではない)，新しいプログラム・カウンタとともに，システム・スタック・ポインタの初期値が入っています．

7.7 不正命令と未実装命令

68000が有効な命令を含まない語を実行しようとすると，トラップが発生し，違反した語を指すプログラム・カウンタ値が保存されます．この場合，3つの例外ベクタのいずれか1つが使用されます．

語の最上位4ビットが1010または1111である場合，その命令は，不正命令というよりもむしろ，**未実装命令**であると見なされます．これらの命令のグループは，68000の将来的なモデルで使用される予定にあるか，あるいは，将来用意される個々の**コ・プロセッサ・チップ**のために使用されます．未実装命令は，これらの4ビットに応じて，2つのベクタのうち1つに対してトラップを発生させます．これはソフトウェアによって未実装命令のエミュレーションを可能にするためです．命令の仕様を与えられれば，その命令とまったく同じ作用をする例外処理ルーチンを書くことができます(ただし命令自体よりもやや遅くなりま

す)．すなわち，未実装命令を含んだプログラム(主に，68000の将来的なバージョン用に作成されたもの)を，修正せずに実行できるソフトウェアを準備することができます．

これらのグループのどちらにも入らない不正命令は，〝不正命令〟ベクタによるトラップを発生させます．このトラップの発生はワイルド・ジャンプ(初期設定されていないアドレス・レジスタを経由するジャンプ)，またはプログラムから飛び出して(はずれて)，データまたは未使用メモリへ入ることの前兆です．

7.8 トラップの原因となる命令

ある種の命令は，その命令の通常の実行の一環として，トラップを発生させます．その理由としては，トラップが命令の主要な働きである場合と，その命令の実行によって何らかの異常状況が発生する可能性がある場合の，2つがあります．

TRAP 命令は常に例外を発生させます．TRAP のオペランドは，0～15の範囲の数であり，それに応じて16の例外処理ベクタのうち1つを使用します．したがって，実際には，16種類の TRAP 命令があることになります．TRAP 命令の主な用途は，オペレーティング･システムまたはモニタに対する呼び出しです．前にも説明したように，保護されたシステムにおいては，オペレーティング・システム・コードをスーパーバイザモードで実行して，ユーザー・プログラム(ユーザーモードで走る)の動作を規制できるようにする必要があります．TRAP 命令によって，プログラムはオペレーティング・システム内のサブルーチンを呼び出すことができ，その呼び出し動作の一環として，プロセッサをスーパーバイザモードにします．呼び出しの種別は，TRAP のオペランド，またはレジスタ内の引数によって伝達することができます．

TRAP のもう1つの用途は，デバッキング･プログラムでブレーク･ポイントをセットするためです．TRAP は2バイト長なので，任意の命令の最初の語と置き換えることができます．プログラムがその点に到達すると，例外処理が発生し，デバッガはユーザーに対するメッセージを表示します．次の章のモニタでは，この方法でブレーク・ポイント機能を実現しています．

この他に特定の条件が真である場合に，トラップを発生させる命令が2つあります．これらの命令は2つとも，プログラムの実行中に発生する可能性のあるエラーを検出するための安価なテストとしての役割を果たします．また，通常は高級言語用のコンパイラによって，自動的に適切な位置に挿入されるものです．TRAPVは，Vコンディション・コードがセットされた場合に，強制的に例外処理を発生させます．プログラムのすべての算術演算のあとにTRAPVを挿入した場合，オーバーフローが生じた箇所で例外処理が発生します．例えば，次のような一連のコードを使用することができます．

```
ADD.L    (A1),D4
TRAPV
ASL.L    #2,D4
TRAPV
```

もう一方の命令がCHKで，この命令は配列に対するアクセスが，その配列の範囲内であることを検査する目的で作られています．CHKは，第1オペランドによって参照される値を，第2オペランドである，データ・レジスタの下位16ビットと比較します．データ・レジスタ内の値が負か，または第1オペランドより大きい場合，トラップが発生します．A1にバイトの配列のアドレスが入り，D1に更新したい配列要素のオフセットが入り，またA2によって指される語には，配列の上限が入るものとすると，CHKは次のように使用することができます．

```
         CHK      (A2),D1         Check that offset is in
*                                 range
         MOVE.B   VALUE,0(A1,D1.W) Update array byte
```

例外処理を強制的に発生させ得るもう2つの命令は，DIVUおよびDIVSの除算命令です．これらは両方とも，0による除算を行おうとした場合，トラップを発生させます．

上記の命令のいずれかが例外処理を発生させた場合，保存されたプログラム・カウンタは，手続き中の次の命令を指した状態になります．

7.9 特権違反

前にも述べたように，ある種の命令は，プロセッサがスーパーバイザモードの場合にのみ実行することができます．これらの命令はいずれも，ステータス・レジスタを更新するか，あるいは周辺装置をリセットすることなどによって，プログラムがオペレーティング・システムからコンピュータの制御を盗めるようにするものです．そのため次の命令は特権化されています．

```
RESET
STOP     #xxxx
RTE
MOVE.W   <ea>,SR
ANDI.W   #word,SR
ORI.W    #word,SR
EORI.W   #word,SR
MOVE.L   USP,An
MOVE.L   An,USP
```

RESET 命令は，プロセッサ・チップからのリセット出力をアサートし，これによってすべての外部装置は初期状態に戻ります．RESET 命令は通常，オペレーティング・システムまたはモニタのスタートアップ時に実行される，最初のいくつかの命令の１つであり，通常の実行においては再び実行されることはありません．この命令はリセット例外処理とは直接つながりはありません．しかし，確実にすべての周辺装置を一定の状態にするための初期化手続きに飛び込む場合には，リセット例外処理ベクタを飛び先として参照することもあります．

STOP 命令は，プロセッサをストップ状態（この状態を，回復不可能なエラーの後でセットされるホルト状態[注6]と混同しないこと）にします．STOP は，次の割込みまたはリセット例外が発生するまで命令の実行を停止します．STOP のオペランドは，16ビットのイミディエイト値であり，ステータス・レジスタに入れられます．これによって，STOP は，コンピュータを停止させる前に，プロセッサの割込み優先順位をセットすることができます．

この命令は，68000を中心に構成されたコンピュータ・システムの周辺装置が，ダイレクト・メモリ・アクセス（DMA）機能を持っている場合に使用するよう意図されています．すなわち，DMA によれば，68000自体に割込みをかけること

なく，周辺装置が直接，メモリに対して読み書きを行えるということを意味します．ディスク装置ではDMAを使って接続するのが一般的であり，メモリとディスクとの間で大量のデータを高速で転送することができます．実行中のプログラムへの割込みは，転送が終了したときにのみ行われます．DMA装置ではしばしば，プロセッサと同時期にメモリの使用を要求する場合が発生するので，この要求を調整し，一方が他方を待つようにするための回路があります．プロセッサがデータのDMA転送を開始させ，転送が終了するまで何もすることが無い場合，プログラムは，割込みが発生するまでループに入ります．しかし，これだとメモリに対する不必要なアクセスを行うことになるので(ループの命令をフェッチするため)，DMA転送の速度が遅くなります(無駄な調停が生じるのでDMAが待たされる)．このように68000がまったく動作しない場合は，STOPを使って停止させた方が良いでしょう．

STOPのオペランドは，ステータス・レジスタのスーパーバイザモード・フラグに対応するビットが1でなければなりません．そうでない場合，STOPがスーパーバイザモードで実行されたとしても，特権違反が発生します(スーパーバイザモードで特権違反が発生するのはこの場合だけです)．したがって，STOP命令の典型的な例を次に示します．

```
STOP    #$2000
```

これは，割込みマスクを0にセットし，どのレベルの割込みも行えるようにします．

USPとの間でのMOVE命令は，特権化する必要はありません．ただし，この命令は，スーパーバイザモードで走っているプログラムでのみ使用するよう，意図されています．というのは，ユーザーモードでは，すでにユーザー・スタック・ポインタはアクセス可能(SPとして)だからです．ユーザーモードでこの命令を使用しようとすると，ユーザーによる処置が必要な，プログラミングエラーが発生する可能性[注7]が強いので，この種の命令は特権化されています．

特権違反トラップが発生したときは，保存されるプログラム・カウンタは，違反した命令を指した状態になっています．

7.10 トレース

各命令の実行後に，内部例外処理を68000に発生させるよう，要求することができます．これがトレース例外処理で，ステータス・レジスタ内のトレース・ビットが1になっている場合に発生します．保存されるプログラム・カウンタは，保存された命令の次の命令を参照します．

トレース例外処理の主な用途は，デバッグ支援です．テスト中のプログラムを，一度に1命令ずつ実行させ，個々の命令を実行するたびに，制御をデバッキング・プログラムに戻します．トレース例外処理は，誤りのあるプログラムの中から，エラー箇所を分離するための強力なツールとなり得ます．次の章のモニタ・プログラムでは，この例外処理を使用したトレース機能を実現しています．

さらにトレース例外処理は，ブレーク・ポイントの取扱いを大幅に単純化します．TRAP命令を使って，ブレーク・ポイント指定された命令の最初の語を置き換える方法は，すでに説明しました．TRAPが発生すると，制御はデバッガに渡され，そこでユーザーはレジスタやメモリ内容を調べるなどの処置を行うことができます．ただし，ブレーク・ポイントに達したあとで，そこからプログラムの実行を続けたい場合は困難な問題が生じます．プログラムがブレーク・ポイントに再び達したとき，そのブレーク・ポイントにはまだ効力を持たせておきたいけれども，しかし，続行するためには，TRAPで置き換える前の元の命令を実行しなければなりません．行うべき処置は，その命令を復元し，それを実行して，次へ進む前にTRAPを元の位置に戻すことです．トレース例外処理を利用することによって，この処置が可能になります．元の命令はトレース・フラグが1の状態で実行されるので，制御がデバッガに返され，デバッガによってブレーク・ポイントTRAPを戻し，次に，トレース・フラグが0の状態で通常の実行を続けることができます．

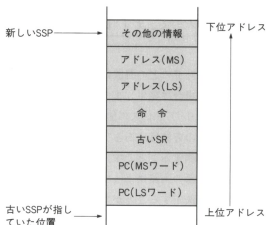
7.11 バスエラーとアドレスエラー

バスエラーは，プロセッサ外部のどの装置にも所属していないアドレスとの間で読み書きを行おうとした場合に発生します．バスエラーは通常，どの物理的メモリにも対応しないメモリ・アドレスの使用を試みたことから発生します．このエラーを検出できる唯一の方法は，あるアドレスが使用された場合に，何の応答も返ってこないのを観測することです．

この検出は，プロセッサ外部の回路によって実行されるので，設計者は，応答に対する適切な待ち時間を決定することができます．もし，このタイム・リミットがプロセッサに組み込まれているなら，ある種の低速な装置を使うことはできなくなるかもしれません．

アドレスエラーは，奇数メモリ・アドレスでワードまたはロングワードのデータを読み書きしようとした場合に発生します．このエラーはバスエラーと非常によく似ていますが，しかし，プロセッサ自体で検出され，異なる例外処理ベクタを使用します．

バスエラーまたはアドレスエラーの原因を正確に判定するのは困難です．現在の命令のオペランド・アドレスが無効である場合も考えられますし，次の命令をプリフェッチする間に発生したのかもしれません．また，ほとんどの例外処理は，命令と命令の間で処理されるか，あるいは1つの命令によって引き起こされるかのどちらかなので，これらのエラーは，命令処理中の任意の時点で検出することができます．すなわち，保存されたプログラム・カウンタの値は，違反した命令の近くの位置を指しているのであり，命令そのものを指しているわけではないということを意味します．

どこが違反しているかを判別できるようにするために，プロセッサは，他の例外に関する情報よりも多くの情報を，スタックに保存します．スタックの合わせて7ワードが使用され，構成は次のようになっています．

命令フィールドには，バスエラーまたはアドレスエラーの発生時に，処理中だった命令の最初のワードが入っています．これによって例外処理ルーチンは，保存されたプログラム・カウンタの値から手前を探索して，メモリの中から命令の先頭部分を見つけ出します．アドレス・フィールドには，アクセスが試みられたアドレスが入っています．最後の語には，打ち切られたバス・サイクルに関する情報が入っています．この形式は，次のとおりです．

Rビットはアクセスが読み出しの場合に1，それ以外の場合は0となります．Nビットは，エラー発生時に68000が命令または命令によって発生した例外処理を処理していなかった場合に，1となります．すなわち，この場合，68000がストップ状態(STOP 命令のあと)であったか，あるいは他の種類の例外処理をすでに行っていた可能性があります[注8]．Fフィールドには，チップからのファンクション・コード出力ラインに乗っていた3ビット値が入っています．ファンクション・コード出力ラインは，アクセスを，スーパーバイザとユーザー，プログラムとデータに分類するラインであり，このラインを使って，メモリを4つの

アドレス空間に分割することができます．

バスエラー，アドレスエラー，またはリセット用の例外処理ベクタに，無効なアドレスまたは奇数アドレスが入っている場合[注9]，例外処理中に，バスエラーまたはアドレスエラーが発生します．このことを〝ダブルバス・フォルト〞といい，回復不可能な障害として取り扱われます．この場合プロセッサは実行をあきらめて自分自身をホルト（停止）状態にし，メモリ内の証拠が破壊されないようにします．この状態から復帰する唯一の方法は，外部リセット信号をプロセッサに送ることです．

一般的に，68000はバスエラーを発生させた命令から実行を続けることはできません．というのは，その命令の内部的な実行の途中である可能性があるからです．68010および68020では，スタックにより多くの情報を保存するので，バスエラーの原因となった命令を継続することができます．

7.12 例外処理の順位付け

複数の例外処理が，同時的に発生する場合が多くあります．このような場合，これらの例外処理が行われる順序を知ることが重要です．各種の例外処理は，それが処理される順序に応じて，3つのグループに分けられます．

グループ0：リセット，バスエラー，アドレスエラー
現在の命令の実行が打ち切られる．

グループ1：トレース，割込み，特権違反，不正命令
現在の命令の実行完了後，次の命令が開始する直前に例外処理が発生します．特権違反および不正命令トラップは，違反した命令の実行の直前に発生します．

グループ2：TRAP，TRAPV，CHK，0による除算
例外処理は，通常の命令実行の一環として発生します．

例外には優先順位があり，一度に複数の例外処理が発生した場合に，どういう処置がとられるかを決定します．最高の優先順位を持った例外処理が最初に処理され，以下昇順で実行されていきます．この順位は，次のとおりです．

1．リセット
2．バスエラー
3．アドレスエラー
4．割込み
5．不正命令，特権違反(同時には起こり得ない)
6．TRAP，TRAPV，CHK，0による除算(同時には起こり得ない)

トレース・フラグがセットされている場合で，現在の命令がリセット，バスエラー，またはアドレスエラーによって打ち切られたとき，トレース例外処理は発生しません．トレースされた命令のあと，割込みが待たされている場合，割込みの前にトレース例外処理が発生します．しかし，現在の命令によって例外処理が発生する場合は，トレース例外処理の前にその例外処理が行われます．

7.13 メモリサイズ判定ルーチン

コンピュータ内の使用可能メモリ量を判定するルーチンで，バスエラー例外処理を使用することができます．この種のルーチンは一般的に，オペレーティング・システムの実行開始時に，自分自身によって実行されます．そうすることによって，異なる記憶容量を持ったコンピュータにも同じオペレーティング・システムをロードすることができ，オペレーティング・システムは常に，使用可能な全メモリを利用することができます．

このルーチンは，メモリを昇順に見ていき，各バイトを順にアクセスしようとします．結果的にこのルーチンは，存在していないバイト(バスエラー例外処理の発生原因となる)をアクセスしようとします．主ループに入る前に，バスエラー例外処理ベクタが，このルーチン中の命令(ラベルSSF_BERR)を指すようセットされます．バスエラー時に記憶される情報は，実際のところ必要ないの

で，スタック・ポインタは，単に元の値にリセットしてしまいます．

メモリ・アドレス・デコーダが，実際にメモリが存在するより，高位の番地に対しても，あたかもそれが存在するように作られている場合(イメージを持つ)には，バスエラー回路ではメモリが実際にあるのかどうかはわかりません．このような状況を防止するために，このルーチンは，各バイトについて，あるビット・パターン書き込み，次にそのパターンがまだ保持されているかどうかを調べることによって，そのバイトが確かにメモリとして機能しているかどうかを調べます．存在しないメモリは，全ビット1または全ビット0になっている可能性があるので，選択されるビットパターンは，1と0の両方を含んでいます．各バイトの元の内容は，レジスタに保存され，テストのあとで復元されるので，メモリの内容は変化しません．テストプログラム自身が常駐しているメモリはテストの対象としないよう，注意しなければなりません．というのは，どこかの位置で，次に実行する命令のバイトを変更してしまうかもしれず，その場合，望ましくない結果が発生するからです．このルーチンは，最後のバイトの直後(SSF_END)からテストを開始します．

```
* Set A0 to the size of available memory in bytes.
*
* The test starts from the end of this routine, and assumes
* that there is one contiguous block of memory.

MEMPAT  EQU     $AA             Pattern used for memory test
I_BERR  EQU     $8              Address of bus error exception
*                               vector

* Plug bus error trap vector to call code here

        MOVE.L  I_BERR,D0       Use D0 to save old trap addr
        MOVEA.L SP,A6           Save old stack pointer
        LEA     SSF_BERR,A0     Address for trap
        MOVE.L  A0,I_BERR       Plug trap vector

* Start the search from the byte after the end of this
* routine

        LEA     SSF_END,A0      Get addr for start of search
```

```
* Main loop of store size finder

SSFLOOP MOVE.B  (A0),D1        Save byte(may cause bus error)
        MOVE.B  #MEMPAT,(A0)   Load pattern
        CMPI.B  #MEMPAT,(A0)   See if there is memory there
        BNE.S   SSF_FOUND      Have hit top of store

        MOVE.B  D1,(A0)+       Restore old contents and go up
*                              one byte
        BRA.S   SSFLOOP        Only exit by branch or bus
*                              error

* Bus error trap comes here

SSF_BERR MOVEA.L A6,SP         Reset system stack pointer

* Memory size found: in A0 in bytes

SSF_FOUND MOVE.L D0,I_BERR     Restore bus error trap address
SSF_END
```

監訳者注

注1： 正確にはこの他に，暗黙的なベクタ番号を使用するものとして，リセット(#0)，バスエラー(#2)，および疑似割込み(#24)の3種類がある．

注2： ファンクション・コード2が，1のときスーパーバイザモード，0のときユーザーモード．

注3： ユーザーモードではA7の参照は常にUSPに対して行われる．

注4： リセット例外処理だけはこれを行なわない．何故なら，SSPの値が不定となるため，スタックへの退避が無意味になる．

注5： つまり，そのような手続きでは，再び呼び出された割込みによって，手続きの環境が変更されてしまい，元の割込み手続きに復帰しても，正しく実行を継続できない場合が生じる．この状況はリエントラントでない手続きの場合とよく似ている．

注6： 2重バス障害が生じたとき，回復不可能なエラーとしてホルト状態になる．

注7： 例えば，ユーザーモードにいるにも関わらず，自分自身がスーパーバイザモードであると勘違いしているプログラムで起こり得るケースである．

注8： 「Nビットは，…… 可能性があります.」とあるのは，より正確には，「Nビットの状態からわかるのは，バスエラー または アドレスエラー が生じた時点で，プロセッサがグループ0，または1の例外処理中(N＝1のとき)か否(N＝0)かを示すにすぎない(グループ0，1についてはP181参照)」というべきである．

注9： さらにこれらに加えて，SSPが奇数値でアドレスエラー例外処理が発生した場合がある．

CHAPTER 8

モニタ・プログラム

8.1 定数の定義 191
8.2 入出力 194
8.3 分岐テーブル 201
8.4 初期設定とコマンド 203
8.5 単純なコマンド・ルーチン 205
8.6 レジスタの表示と更新 207
8.7 ユーザー・プログラムの実行 212
8.8 メモリの確認・更新ルーチン 216
8.9 ブレーク・ポイント 220
8.10 例外処理手続き 222
8.11 メッセージとテーブル 229

はじめに

本章では完全なプログラム例を示してます．このプログラムは小型のモニタで，機械語の実行およびデバッグのための，限定された範囲での機能を提供します．一般的にこのようなモニタは，ROM に常駐しており，ここで示すモニタよりも，はるかに多くのコマンドが用意されています．本章で説明するモニタは限定されてものではありますが，特に68000の割込みとトラップ・ベクタの使用法，スーパーバイザモードとユーザーモードの使用法を示しています．

このモニタは，メモリ内の任意の番地に常駐できるようになっており，コードがすべて確実に位置独立型となるように注意が払われています．モニタ自体はスーパーバイザモードで走りますが，ユーザー・プログラムは，ユーザーモードでのみ実行することができます．

また，このモニタには次のコマンドが使用されています．ユーザーのレジスタ・セットの確認および変更，ブレーク・ポイントのセットおよびクリア，メモリ内容の確認および変更，ユーザー・プログラムの通常モードまたはトレースモードによる実行などです．

ユーザー・レジスタの値は，R コマンドによって表示することができます．このコマンドは全レジスタの値を表示しますが，特定のアドレス・レジスタまたはデータ・レジスタの値を表示するには，A または D コマンドを使用します．

同様に，P および S コマンドを使って，プログラム・カウンタおよびステータス・レジスタを選択的に表示することができます．

A, D, P, または S コマンドのあとに16進数値を指定することにより，任意のレジスタの内容を更新することができます．

また，M コマンドを使ってメモリ番地をオープンすることができます．オープンした場合，M のあとに指定した番地のメモリ内容が表示されます．

このような方法によってメモリ番地をオープンしたあとでは，メモリ・サブコマンドを使用します．メモリ番地は，初期的にはバイト値としてオープンされますが，W または L を入力することによって，そのアドレスで始まるワード(2バイト)またはロングワード(4バイト)を指定することができます．このとき，奇数番地のバイトは，ワードまたはロングワードとしてオープンすることはで

きません．

モニタ・コマンド	
Ar Ar n	ユーザー・アドレス・レジスタｒの値を表示する ユーザー・アドレス・レジスタｒを値ｎに更新する
B Bi Bi n	現在のブレーク・ポイントを表示する ブレーク・ポイントｉをクリアする ブレーク・ポイントｉを番地ｎにセットする
C	ブレーク・ポイントの後から継続する
Dr Dr n	ユーザー・データ・レジスタｒの値を表示する ユーザー・データ・レジスタｒを値ｎに更新する
G G n	ユーザー・プログラムを，現在のユーザー・プログラム・カウンタで開始する ユーザー・プログラムを，アドレスｎで開始する
M n	メモリ番地ｎをオープンし，メモリ・コマンドを処理する
P P n	ユーザー・プログラム・カウンタの値を表示する ユーザー・プログラム・カウンタを，値ｎに更新する
R	すべてのユーザー・レジスタの値を表示する
S S n	ユーザー・ステータス・レジスタの値を表示する ユーザー・ステータス・レジスタを，値ｎに更新する
T T n	ユーザー・プログラムを，ユーザー・プログラム・カウンタからトレースする ユーザー・プログラムを，アドレスｎからトレースする

このようにしてメモリ番地をオープンしたあと，新しい値を16進数で入力することができます．この値によって，現在選択されているサイズに応じて，4，2または1バイトが置き換えられます．上向き矢印(∧)は，現在選択されているサイズの前の番地をオープンするのに対し，〝リターン〞は，次の番地をオープンします．等号(＝)は，現在の番地を再び表示するのに使用しますが，これは，メモリ・アドレスが実際にはメモリ・マップされたＩ/Ｏ番地である場合に便利です．メモリ・サブコマンド・レベルは，フルストップ(．)によって終了し，通常のモニタに戻ります．

ユーザー・プログラムをメモリに入れるには，機械語の16進表現を使って，Ｍコマンドによって行います．より完全なモニタでは，他のコンピュータから

プログラムをロードする機能も備わっています．

メモリ・コマンド	
リターン	次のメモリ番地へ移動する
^	前のメモリ番地へ移動する
=	現在の番地を再び表示する
.	メモリ更新コマンドから出る
n	現在の番地を値nに更新する
L	ロング値(4バイト)として表示する
S	バイト値(1バイト)として表示する
W	ワード値(2バイト)として表示する

プログラムをメモリに入れたら，Gコマンドによって実行することができます．これによって，コンピュータはユーザーモードに入り，ユーザー・プログラム・カウンタで指定している番地へジャンプします．もし必要ならば，プログラム・カウンタの新しい値を，Gコマンドのあとに指定することができます．

プログラムをデバッグするためにTコマンドを使って実行を開始することもできます．この場合，1つの命令が実行されると，再びモニタに制御が戻されます．レジスタ値が表示され，モニタ・コマンド待ちとなります．だたし，この段階で〝リターン〟だけを入力した場合，次の命令が実行されます．これによって，簡単にプログラムを1ステップごとにトレースすることができます．

さらにモニタによって，ユーザー・プログラム内にブレーク・ポイントをセットすることができます．Bコマンドだけの場合は，現在のブレーク・ポイントをすべて表示します．Bnは，ブレーク・ポイントnを削除し，Bnのあとに16進数を指定すると，指定したアドレスにブレーク・ポイントnをセットします．ブレーク・ポイントが検出されるとモニタに戻ります．Tコマンドは，複数の命令をトレースするために使用することができ，ブレーク・ポイントから継続するにはCコマンドを使用します．

モニタではすべての例外処理とトラップが処理されます．TRAP #15はユーザー・プログラムの終了を示すのに使用し，TRAP #14はブレーク・ポイントを発生させるのに使用します．トレース・ベクタは，ユーザー・プログラムのトレース中に使用されます．その他のすべてのトラップまたは例外処理に関しては，

適切なメッセージが書き出され，プログラムはレジスタの内容を保存した状態で停止します．ブレークまたはトレースのあと，レジスタの内容が表示されます．さらに，トレース例外処理が発生したあと〝リターン〞と応答することによって，次の命令がトレースされます．

使用される唯一の割込み番地は，レベル2自動ベクタ番地です．このベクタをACIAが割込みで使うものと見なし，ACIAの番地は，プログラム内でロング値としてとられるので，これにパッチをあてて，メモリ・マップにおけるACIAの実際の番地を変えることができます．同様に，スタックおよび他のデータ領域用にモニタが使用するRAMの2つの領域は，プログラム内の他の2つのロング値によって定義されます．

8.1 定数の定義

まず最初に，使用する各種の定数を定義し，RAM番地の割付けを指定しなければなりません．最初のセクションでは，各種のASCII定数，ステータス・レジスタ内のビット，およびいくつかのデフォルト値を定義します．さらに，ブレーク・ポイントとして使用される，TRAP #14に対する2進オペコードを指定します．

```
SPACE    EQU    $20      ASCII space character
CR       EQU    $0D      ASCII carriage return
LF       EQU    $0A      ASCII line feed
BS       EQU    $08      ASCII backspace
DEL      EQU    $7F      ASCII delete
TBIT     EQU    7        Trace bit in saved status
*                        register
SBIT     EQU    5        Supervisor bit in it
INTSOFF  EQU    $2700    Interrupts off
INTSON   EQU    $2000    Interrupts on
ISTK     EQU    $4000    Initial stack pointer
INTVECS  EQU    0        Location of fixed vectors
BRKTRP   EQU    $4E4E    TRAP 14 instruction
DEFSP    EQU    $2000    Default USP
```

次のセクションでは，使用される RAM 領域の構造を定義します．作業領域として固定的な番地は使用せず，その代わりに常にアドレス・レジスタ(通常，A6)からのオフセットを使用します．打ち込まれる入力文字を取り扱うために，ライン・バッファへの 3 個のポインタと，バッファ自体が必要です．さらに，ユーザーのレジスタとブレーク・ポイント番地用の空間を割付けます．また，ユーザーのレジスタ保存領域のオフセットに対して名前を付けます．

```
BUFWR    EQU    0              Write pointer
BUFRD    EQU    BUFWR+4        Read pointer
BUFLS    EQU    BUFRD+4        Line start pointer
BUFBEG   EQU    BUFLS+4        Buffer area
BUFEND   EQU    BUFBEG+79      End of buffer
RDUMPD   EQU    BUFEND+1       Space for 8 data registers
RDUMPA   EQU    RDUMPD+32      Space for 7 address registers
RDUMPSP  EQU    RDUMPA+28      Space for USP
RDUMPSR  EQU    RDUMPSP+4      Space for user status register
RDUMPPC  EQU    RDUMPSR+2      Space for user program counter
BRKP     EQU    RDUMPPC+4      10 breakpoints, 6 bytes each
BFLG     EQU    BRKP+60        Space for breakpoint flag
RAMEND   EQU    BFLG+2         End of RAM area

D_A0     EQU    RDUMPA-RDUMPD Offset of register A0
D_A7     EQU    RDUMPSP-RDUMPD Offset of register A7
D_SR     EQU    RDUMPSR-RDUMPD Offset of status register
D_PC     EQU    RDUMPPC-RDUMPD Offset of user program
*                              counter
RDSIZE   EQU    (RAMEND-RDUMPD)/4 Size of save area in long
*                              words
```

すべての定義が終わったら，次にいくつかの定数領域の割付けを行います．この最初の領域は，例外処理，割込み，およびトラップ手続きのエントリを定義します．コードが 0 番地から始まるようロードされる場合，例外処理ベクタは，プログラムの先頭に置かれます．この場合，コンピュータのハードウェアがプログラムとデータに対して，異なるメモリ領域を提供していないとき，例外処理ベクタは正しい位置にあることになります．プログラムが他の場所にロードされている場合は，モニタのこの初期設定セクションは，これらの番地を番地 0 で始まるメモリにコピーします．最初の 2 つのロング番地は，68000のリセット時に取られる，スタック・ポインタおよびプログラム・カウンタの初期値を定義します．正常にリセットが働くためには，モニタ全体が 0 番地からロードされるか，あるいは，最初の 8 バイト(0〜7 番地)がハードウェアによって

モニタ内の最初のアドレスにマップされなければなりません．後者の場合，開始アドレスは，適切な方法でリロケートされる必要があります．

```
TS       EQU     *
I_RESET  DC.L    ISTK          RESET stack pointer
         DC.L    START         RESET program counter
         DC.L    B_EXCPT-TS    Bus error
         DC.L    A_EXCPT-TS    Address error
         DC.L    I_EXCPT-TS    Illegal instruction
         DC.L    D_EXCPT-TS    Divide by zero
         DC.L    C_EXCPT-TS    CHK exception
         DC.L    O_EXCPT-TS    TRAPV exception
         DC.L    P_EXCPT-TS    Privilege exception
I_TRACE  DC.L    T_EXCPT-TS    Trace exception
         DC.L    X_EXCPT-TS    L1010 emulation
*                              (illegal instruction)
         DC.L    Y_EXCPT-TS    L1111 emulation
*                              (illegal instruction)
         DS.L    12            Unassigned as yet
         DC.L    S_EXCPT-TS    Spurious interrupt
*
* Autovectored interrupts
*
         DC.L    INT-TS        Interrupt level 1
         DC.L    CINT-TS       Interrupt level 2 (console)
         DC.L    INT-TS        Interrupt level 3
         DC.L    INT-TS        Interrupt level 4
         DC.L    INT-TS        Interrupt level 5
I_INT6   DC.L    INT-TS        Interrupt level 6
I_INT7   DC.L    INT7-TS       Interrupt level 7
*
* Trap vectors
*
         DC.L    TRP-TS        Unexpected TRAP
         DC.L    TRP-TS
         DC.L    TRP-TS
         DC.L    TRP-TS
         DC.L    TRP-TS
         DC.L    TRP-TS
         DC.L    TRP-TS
         DC.L    TRP-TS
         DC.L    TRP-TS
         DC.L    TRP-TS
         DC.L    TRP-TS
         DC.L    TRP-TS
         DC.L    TRP-TS
         DC.L    TRP-TS
         DC.L    TRP14-TS      Breakpoints
I_BRK    DC.L    TRP15-TS      User requests
*
INTSIZE  EQU     (I_BRK-I_RESET)/4 Size of fixed vectors
```

最後に，ACIAと使用するRAM領域の基底番地を指定します．これは，プログラム全体をアセンブルし直すことなく修正できるようにプログラム空間に入れられます．

```
ACIA     DC.L    $83FF01          Address of ACIA
RAMBASE DC.L     $1000            RAM base pointer
```

8.2 入出力

これでコードを正しく開始できるようになりました．最初のいくつかのサブルーチンは，任意のユーザー・プログラムから呼び出せるようになっており，よって，どのレジスタも特定の値を持たないものと見なされます．各種のモニタ・コマンドを取り扱うあとのサブルーチンでは，最後に打ち込まれた文字はレジスタD0に入り，レジスタA6はRAM領域の基底を指すものと見なされます．

最初に定義するルーチンは割込みルーチンです．ルーチンの先頭のアドレスは，割込みレベル2に対する自動ベクタに置かれるので，このルーチンは，キーボードから文字が打ち込まれたら，常に呼び出されます．このルーチンは，文字を取り込み，それをリング・バッファにいれます．バッファがいっぱいでそれが不可能な場合は，その文字は単に無視されます．文字が**ラブアウト**(1文字削除)である場合，現在の行にそれ以上文字が残っていない場合を除いて，最後に打ち込まれた文字が取り除かれます．

一般的に，打ち込まれた文字はターミナル上に反映されます．〝ラブアウト〟が打ち込まれた場合は，バックスペース，スペース，バックスペースが行われ，ターミナル・スクリーンから文字が消されます．〝リターン〟が打ち込まれた場合は〝リターン〟に続いてラインフィードが行われます．

割込みルーチンは，あらゆる時点で呼び出される可能性があるので，割込みルーチンで使用されるレジスタを保存するのは，重要なことはもちろんです．また，RTE命令によって，保存されていたステータス・レジスタの内容が，復元されることにも注意してください．

リング・バッファは，3つのポインタによって管理されます．BUFWRは現在

の書込み位置として使用され，バッファに最後に入力された文字を指します．通常の場合，このポインタはサブルーチン **INCPTR**（ポインタを適切な値に更新する）によってインクリメントされます．ポインタ **BUFRD** は，最後に読み出した文字位置を示します．**BUFWR** が **BUFRD** に達した場合，バッファ内にはもう新たに文字を追加する余地がありません．読取りルーチンは，いったん行全体が打ち込まれてから，バッファから文字を取り出すだけです．したがって，行を編集することができます．ポインタ **BUFLS** は現在の行の先頭を指すように設定され，バッファの状態をチェックするために使われます（空のバッファからの読み出し，すでにいっぱいなバッファへの書き込みなど）．そのため，行全体がすでに抹消されている場合は，ラブアウトは無視されます．ラブアウトが打ち込まれると，**BUFWR** ポインタはそれごとにデクリメントされます．

文字の反映は，ルーチン **WRCH** によって取り扱われます．このルーチンは D0 に入っている文字を，ACIA の出力部，すなわちターミナルへ表示します．出力要求が完了した時点で，ACIA が割込みをかけるよう構成することもできますが，それはこの例では使用されていません．入力文字が到着した時点でのみ，割込みが発生するようになっています．このモニタは，一度に 1 つのプログラムを走らせるように作られているので，ACIA が伝送を完了するのを待っている間は，何の動作もしません．受信割込みを使用することは，文字のタイプアヘッドができることを意味しています．

```
*
* Console interrupt routine
*
CINT    MOVEM.L D0/A1/A2/A6,-(SP) Save registers
        MOVEA.L ACIA,A1          Get address of ACIA
        MOVE.B  2(A1),D0         Get character and cancel
*                                interrupt
        ANDI.B  #$7F,D0          Strip parity bit
        MOVEA.L RAMBASE,A6       Establish pointer to base area
        MOVEA.L BUFWR(A6),A2     Get write pointer
        CMP.B   #DEL,D0          Is this delete?
        BNE.S   CINT2            No .. handle normal character
        CMPA.L  BUFLS(A6),A2     Start of line?
        BEQ.S   CINT4            Yes .. nothing to do
        LEA.L   BUFBEG(A6),A1    Get pointer to buffer start
        CMPA.L  A1,A2            If equal then cyclic decrement
        BNE.S   CINT1            No .. normal decrement
        LEA.L   BUFEND(A6),A2    Yes .. set to buffer end
CINT1   SUBQ.L  #1,A2            Decrement pointer
        MOVEA.L ACIA,A1          Restore ACIA pointer into A1
```

```
        MOVEQ   #BS,D0          Get backspace into D0
        BSR.S   WRCH            And send it
        MOVEQ   #SPACE,D0       Next a space
        BSR.S   WRCH            And send it
        MOVEQ   #BS,D0          Finally another backspace
        BRA.S   CINT3           Send it after updating write
*                               pointer
CINT2   BSR.S   INCPTR          Update pointer handling circular
*                               list
        CMPA.L  BUFRD(A6),A2    Check equal to read pointer
        BEQ.S   CINT4           Equal, so no room in buffer
        MOVE.B  D0,(A2)         Store character
CINT3   MOVE.L  A2,BUFWR(A6)    Store write pointer back again
        BSR.S   WRCH1           Write character in D0 to ACIA
*                               in A1
        CMPI.B  #CR,D0          Is this a return?
        BNE.S   CINT4           No .. nothing else to do
        MOVE.L  A2,BUFLS(A6)    Update line start pointer
        MOVEQ   #LF,D0          Place line feed code in D0
        BSR.S   WRCH1           And display it
CINT4   MOVEM.L (SP)+,D0/A1/A2/A6 Restore registers
        RTE                     And return from interrupt
```

このサブルーチンでは，A2にリング・バッファに対するポインタが入っており，A6がRAM領域の基底を指すものと見なされます．このサブルーチンはA2をインクリメントし，終わりに達したらバッファの先頭にリセットします．

```
INCPTR  MOVE.L  A1,-(SP)        Save register
        ADDQ.L  #1,A2           Increment pointer
        LEA.L   BUFEND(A6),A1   Get pointer to end of
*                               area
        CMPA.L  A1,A2           Check if equal
        BNE.S   INC1            No, so ok
        LEA.L   BUFBEG(A6),A2   Reset pointer to start of
*                               buffer
INC1    MOVE.L  (SP)+,A1        Restore register
        RTS
```

次のルーチンは出力に関連するものです．最初のルーチンはWRCHで，D0に入っていた文字をACIAの出力部に伝送します．WRCHは補助ルーチンWRCH1を使用します．このルーチンはA1がACIAを指すものと見なします．

```
WRCH    MOVE.L  A1,-(SP)        Save register
        MOVEA.L ACIA,A1         Extract ACIA address
        BSR.S   WRCH1           Transmit character
        MOVE.L  (SP)+,A1        Restore A1
        RTS
```

ルーチン WRCH1 は，WRCH と WRITES によって，A1 が ACIA コントロール・レジスタを指すよう設定されたあとに呼び出されます．ACIA のデータ・レジスタは，メモリ内で 2 バイト高い位置にあります．WRCH1 は，単に ACIA がレディ状態になるのを待ち，その後，D0 に入っている文字を伝送します．

```
WRCH1    BTST     #1,(A1)       Check if ACIA is ready for
*                               output
         BEQ.S    WRCH1         No, wait until it is
         MOVE.B   D0,2(A1)      Transmit character
         RTS
```

出力処理に関した便利なサブルーチンをいくつか次に示します．BLANK は，出力にスペースを書き出し，NEWLINE は，キャリッジ・リターンに続いてラインフィードを書き出します．

```
BLANK    MOVE.L   D0,-(SP)      Save D0
         MOVEQ    #SPACE,D0     Space code
         BSR.S    WRCH          Write it
         BRA.S    NEWL2         Jump to shared code
*
* Write out a CR, LF to the output
*
NEWLINE  MOVE.L   D0,-(SP)      Save D0
         MOVEQ    #CR,D0        Print carriage return
         BSR.S    WRCH
         MOVEQ    #LF,D0        Print line feed
         BSR.S    WRCH
NEWL2    MOVE.L   (SP)+,D0      Restore D0
         RTS
```

これは，すでに説明したルーチンの変形です．バイト値 0 で終結した何らかの文字例を A0 が指した状態で，WRITES が呼び出されます．文字はすべて，出力に書き出されます．WRITES は，文字例の表示開始時に，1 回だけポインタを ACIA に設定するので，WRCH ではなく，WRCH1 を呼び出します．

最後に NEWLINE に分岐して，文字例の終わりのニューラインを表示します．これは，一般的なトリックを表すもので，サブルーチンの最後の動作が，他のルーチンを呼び出し，続いて RTS 命令を実行することであるなら，直接そのルーチンへ分岐した方が簡単です．WRITES に対する戻り番地は，まだスタック上にあります．そのため，NEWLINE が最終的に自分自身の RTS を実行すると

き，WRITESの呼出し元に，ジャンプして戻ります．

```
WRITES  MOVEM.L D0/A0-A1,-(SP) Save registers
        MOVEA.L ACIA,A1        Extract ACIA address
WRITES1 MOVE.B  (A0)+,D0       Extract character from
*                              string
        BEQ.S   WRITES2        Zero - end of string
        BSR.S   WRCH1          Write out character using
*                              ACIA in A1
        BRA.S   WRITES1
WRITES2 MOVEM.L (SP)+,D0/A0-A1 Restore registers
        BRA.S   NEWLINE        Print newline and return
```

最後に示す一連の出力ルーチンは，16進数を表示します．WRHEX4は4バイト，WRHEX2は2バイト，そしてWRHEX1は1バイトの16進数を表示します．最後のWRHEX0は4ビット(ニブル)を表示します．これらのルーチンはいずれも個別に呼び出すことができますが，すべてWRHEX0を必要な回数だけ呼び出します．この呼び出しでレジスタ値があちこちやり取りされても，破壊されることはありません．

WRHEX4は，レジスタの上下半分をスワップし，WRHEX2の呼び出しを通じて上位の2バイトを表示します．次にWRHEX2のコードに入って，下位の2バイトを表示します．

```
WRHEX4  SWAP    D0             Swap high and low halves
        BSR.S   WRHEX2         Write high 2 bytes
        SWAP    D0             Swap again
* Drop through to WRHEX2
```

WRHEX2も，同様のトリックを実行します．これは下位ワードをローテイトして，対になっている上位バイトを，WRHEX1の呼び出しによって表示します．次にこれをローテイトして戻し，レジスタを復元します．次にWRHEX1のコードに入って，下位バイトを表示します．

```
WRHEX2  ROR.W   #8,D0          Shift top byte down to low
*                              order
        BSR.S   WRHEX1         Write single byte
        ROL.W   #8,D0          Shift bottom byte back
* .. and drop into WRHEX1 for this byte
```

WRHEX1 は，WRHEX2 に非常によく似ています．この場合では，最下位の 4 ビットを下へローテイトし，WRHEX0 によって表示して，次に元へ戻って最下位の 4 ビットを表示します．

```
WRHEX1  ROR.B    #4,D0          Shift down top nibble
        BSR.S    WRHEX0         Write it out
        ROL.B    #4,D0          Put back bottom nibble
*    .. and drop into WRHEX0
```

最後に WRHEX0は，D0に入っている 1 桁の16進数を書き出します．この場合，D0を破壊しないよう注意が払われ，出力を行うために WRCH を呼び出します．

```
WRHEX0  MOVE.L   D0,-(SP)       Save register
        ANDI.B   #$0F,D0        Mask to bottom 4 bits
        ADDI.B   #'0',D0        Add character zero
        CMPI.B   #'9',D0        Test to see if greater
*                               than character 9
        BLS.S    WRHEX01        Just write it
        ADDI.B   #'A'-'9'-1,D0 Convert to character
WRHEX01 BSR.S    WRCH           Write out hex character
        MOVE.L   (SP)+,D0       Restore register
        RTS                     And return
```

このセクションの最後の部分は，入力ルーチンに関連するものです．WRCH の反対の作用をするルーチン RDCH は，ターミナルからの文字をレジスタ D0 に返します．D0 の上位 3 バイトをクリアし，その文字を下位バイトに返すのが便利でしょう．

ここで，リング・バッファが割込みルーチンよって管理されていること，および 2 つのポインタ BUFRD および BUFLS が，それぞれバッファから最後に読み出された文字と，現在の行の先頭を指していることを思い出してください．読出しポインタ BUFRD と，行の先頭ポインタ BUFLS を比較し，もしこれらが等しければ，等しくなくなるまでループを繰返します．ユーザーが行の入力を完了したことを示す〝リターン〟をキーボードから打ち込んだ場合，もちろん，2 つのポインタは等しくなくなります．文字が前もって打ち込まれている場合は待つ必要はなく，一度に取り出すことができます．さらに同じルーチン INCPTR を呼び出してポインタを一個ずつ進めます．

```
RDCH     MOVEM.L A2/A6,-(SP)  Save registers
         MOVEA.L RAMBASE,A6   Pointer to data area
         MOVEA.L BUFRD(A6),A2 Extract buff read pointer
RDCH1    CMPA.L  BUFLS(A6),A2 Equal to line start?
         BEQ.S   RDCH1        Wait until it is not
         BSR     INCPTR       Increment buff read pointer
         MOVEQ   #0,D0        Clear all of D0
         MOVE.B  (A2),D0      Extract character
         MOVE.L  A2,BUFRD(A6) Update buffer read pointer
         MOVEM.L (SP)+,A2/A6  Restore registers used
         RTS
```

このルーチンはキーボードからの16進数を読み込むもので，2つのエントリ・ポイントが準備されています．READHEXは入力からの次の文字を読み，READHは，次の文字がすでに読み込まれ，レジスタD0に入っているものと見なします．結果はD1に戻され，D0はルーチンによって読み込まれた最後の文字にセットされます．無効な文字が発見されるとZフラグはクリアされ，文字が有効な場合はZフラグはセットされます．このため，Zフラグをあとの段階でBNEによってテストして，何らかのエラー処理手続きへジャンプすることができます．

```
READHEX BSR.S    RDCH         Get character
READH   CMPI.B   #SPACE,D0    Check if space
        BEQ.S    READHEX      Discard leading space
        CLR.L    D1           Clear result register
RDH1    CMPI.B   #'0',D0      Check if below character 0
        BCS.S    RDH4         Error exit with Z unset
        CMPI.B   #'9',D0      Check if above character 9
        BHI.S    RDH2         Possibly A .. F
        SUBI.B   #'0',D0      Subtract character 0
        BRA.S    RDH3         And assemble in D1
RDH2    BSR      LOCASE       Convert to lower case
        CMPI.B   #'a',D0      Check if below character a
        BCS.S    RDH4         Error exit with Z clear
        CMPI.B   #'f',D0      Check if above character f
        BHI.S    RDH4         Error exit with Z clear
        SUBI.B   #'a'-10,D0   Convert to correct value
RDH3    ASL.L    #4,D1        Multiply current sum by 16
        ADD.L    D0,D1        Add in this term
        BSR.S    RDCH         Get next character
        CMPI.B   #CR,D0       See if equal to CR
        BEQ.S    RDH4         Yes .. exit with Z set
        CMPI.B   #SPACE,D0    See if equal to space
        BNE.S    RDH1         No.. go back and handle it
RDH4    RTS                   Exit with Z set if all ok
```

8.3 分岐テーブル

モニタ内で入力からコマンドの文字を取り，打ち込まれた文字に基づいて，何らかの動作を決定したい場合があります．この処理を行うために，ルーチンSEARCHを使用します．このルーチンは，D0に文字が，そしてレジスタA0に分岐テーブルへのポインタが入っているものと見なします．

分岐テーブルは，個々の有効な文字に対して必要とされる動作を示すとともに，文字が無効な場合のデフォルト動作を示します．テーブルの各項目は，4バイトから構成されています．最初のバイトはフラグであり，テーブル内にまだ項目がある場合は0，項目がない場合は0以外の値にセットされます．後者の場合，テーブル内の項目は，必要とされるデフォルトの動作を表すものと見なされます．

各項目の2番目のバイトは，コマンド文字が入っています．このバイトは，フラグ・バイトがセットされているときは無視されます．

最後の2バイトは，実行すべき動作を示してます．個々の動作について，関連するサブルーチンがあり，このサブルーチンのアドレスが，この2バイト・スロットで示されます．位置独立のコードを保つため，テーブルの項目は，テーブルの基底からのサブルーチンへのオフセットを表しています．

分岐テーブルを使用する理由には2つあります．まず第1に，同じコードを使って，異なる環境でも有効な，異なるコマンドセットをデコードすることができます．この場合，通常のコマンドおよびメモリ・チェンジ・コマンドをデコードするために，SEARCHを使用します．第2に，テーブルに新しい項目を入れ，その仕事を行うサブルーチンを用意するだけで，新しいコマンドを簡単に追加することができます．

ルーチンSEARCHに対して，文字がレジスタD0(ルーチンLOCASEによって小文字に変換されている)で渡されます．この文字と一致するかどうか，テーブル内の各項目が調べられます．もしあれば，関連するルーチンが呼び出されます．テーブルの終わりにある0以外のフラグ・バイトが検出された場合，デフォルト・ルーチンが常に呼び出されます．

SEARCHルーチンは，いずれのレジスタも破壊しません．レジスタの保存の

ため，スタックを作業領域として使用します．まず，2番目のスタック・フレーム・スロットから読みだすことによって，D0の元の値が復元されます．その後，このスロットは，呼び出されるべきサブルーチンのアドレスによって更新されます．そして，レジスタA0は復元され，スタック・ポインタは必要なエントリ・ポイントおよびSEARCHの呼出し元を示す戻り番地を指すように下げられます．最後のRTS命令はこの値をスタックから取り，必要なサブルーチンにジャンプします．このサブルーチンがRTSを実行すると，SEARCHの呼出し元へ戻ります．

```
SEARCH  MOVE.L   D0,-(SP)     Save register D0
        MOVE.L   A0,-(SP)     and register A0
        BSR.S    LOCASE       Convert to lower case
SRCH1   TST.B    (A0)+        Check if end and skip byte
        BNE.S    SRCH2        Non zero - end of table
        CMP.B    (A0)+,D0     Compare char and skip byte
        BEQ.S    SRCH3        Found it!
        ADDQ.L   #2,A0        Skip routine offset
        BRA.S    SRCH1        And try again
SRCH2   ADDQ.L   #1,A0        Skip unused character byte
SRCH3   MOVEA.W  (A0),A0      Offset of routine from
*                             table base
        ADDA.L   (SP),A0      Add in saved table base
        MOVE.L   4(SP),D0     Restore register D0
        MOVE.L   A0,4(SP)     And replace with routine
*                             address
        MOVE.L   (SP)+,A0     Restore register A0
        RTS                   Pickup routine address and
*                             jump to it
```

次の小さいルーチンは，レジスタD0の中の文字を必要に応じて小文字に変換します．変換を必要とする文字を確定したら，小文字と大文字の差を表す値を加算します．

```
LOCASE  CMPI.B   #'A',D0      Check if alphabetic char
        BCS.S    LOC1         No need to convert unless
*                             it is
        CMPI.B   #'Z',D0      Check again
        BHI.S    LOC1         Still no need
        ADDI.B   #'a'-'A',D0  Convert char to lower case
LOC1    RTS                   And return
```

8.4 初期設定とコマンド

次のコード・セクションは，モニタの初期設定セクションで，ラベルSTARTがプログラム全体のエントリ・ポイントです．このアドレスはリセット・ベクタに入れてあるので，マシンに電源が投入された場合やリセット時には，このプログラムが呼び出されます．

割込み処理手続きは，まだ定義されておらず，割込みが発生すると面倒なので，最初の動作は，割込みをオフにすることです，次に，RESET命令を指定して，このエントリ・ポイントに単純にジャンプが行われた場合に，外部リセットをシミュレートできるようにします．次に，68000のリセットに影響を受けないACIAをリセットします．

```
START   MOVE.W  #INTSOFF,SR   Interrupts off,
*                             supervisor mode
        RESET                 Issue RESET command
        MOVEA.L ACIA,A3       Point to ACIA
        MOVE.B  #$03,(A3)     Reset ACIA
```

次のステップは，割込み処理手続きをRAM内の定義されたスロットにコピーすることです．プログラムが0番地からロードされた場合，それはともかく正しい位置にあることになりますが，プログラムを自分自身の先頭にコピーして戻しても，何も支障はありません．ただしこれは，プログラムがROMに書き込まれており，かつROMに対する書込み動作でバスエラーを生じるようにハードウェアが設定されていない限りにおいてです．ほとんどのハードウェアの構成では，このような場合はありません．

さらに位置の独立性を保つために，テーブルに記憶されている値は，実際のアドレスではなく，プログラムの基底からのオフセットとなっています．前にも述べたとおり，プログラムが0番地からロードされている場合は，これは正しい値ですが，そうでない場合は，プログラムの基底アドレスを個々のオフセットに加算して，正しいアドレスを算出します．

```
         LEA.L     INTVECS,A1      Point to interrupt vector
*                                  area
         LEA.L     I_RESET,A2      Point to defined handlers
         MOVE.L    A2,D2           Maintain base pointer
         MOVE.W    #INTSIZE,D0     Number of slots
STO      MOVE.L    (A2)+,D1        Extract handler location
         ADD.L     D2,D1           Add table base
         MOVE.L    D1,(A1)+        Install in low RAM
         DBRA      D0,STO          Loop until complete
```

次のステップは，システム・スタック・ポインタをリセット・ベクタに記憶された値に設定することです．前にも述べたとおり，外部リセットによってエントリ・ポイントに入ったのではなく，ジャンプされた場合にのみ，これが必要です．この処理が完了したら，再び割込みをオンにしても安全です．

```
         MOVE.L    I_RESET,SP      Initial system stack
         MOVE.W    #INTSON,SR      Interrupts on again
```

これでACIAを初期設定して，ACIAの割込み処理手続きによって使用されるリング・バッファに，ポインタの正しい初期値を設定することができます．さらに，モニタで使用するRAMの基底番地ポインタとして，A6を設定します．この値はモニタの実行中にA6に保持され続けているものと見なされます．

```
         MOVEA.L RAMBASE,A6      Establish RAM address
*                                register
         LEA.L   BUFBEG(A6),A1   Point to buffer start
         MOVE.L  A1,BUFWR(A6)    Initial buffer write
*                                pointer
         MOVE.L  A1,BUFRD(A6)    Initial buffer read pointer
         MOVE.L  A1,BUFLS(A6)    Initial buffer line start
*                                pointer
         MOVE.B  #$89,(A3)       Magic value.
*                                Rx interrupts on.
```

次の段階は，ブレーク・ポイントとユーザー・レジスタ記憶域を0にクリアすることです．カウンタが−1のときにDBRAが停止するので，スロット数より1だけ小さい値をカウンタとして使用します．さらに，ユーザー・スタック・ポインタの初期値を設定します．

```
          LEA.L   RDUMPD(A6),A1  Point to data save areas
          MOVE.W  #RDSIZE-1,D0   Size of area to clear
CL        CLR.L   (A1)+          Clear data area
          DBRA    D0,CL          Branch until done
          MOVE.L  #DEFSP,RDUMPSP(A6)  Set initial USP
```

次に，ヘッダ・メッセージを書き出します．このため，まずA0に，メッセージに対するポインタをロードしてから，WRITESを呼び出します．

```
          LEA.L   MESS1,A0       Point to header message
          BSR     WRITES         Write message
```

これが主実行ループで非常に簡単です．まず，WRCHを呼び出し，プロンプトを書き出します．次にA0に，コマンド・テーブルに対するポインタをロードします．サブルーチンSEARCHが次に呼び出され，このサブルーチンが最終的に，正しいルーチンを呼び出します．そのルーチンから戻った場合，次のコマンドを実行するため，主ループの始めに戻ります．コマンドでエラーが発生したり，あるいはユーザー・プログラムへ入った場合には，例外処理セクションから直接，ラベルST1へジャンプします．

```
ST1       MOVE.B  #'#',D0        Write prompt
          BSR     WRCH
ST2       BSR     RDCH           Get character into D0
          LEA.L   COMTAB,A0      Point to command search
*                                table
          BSR     SEARCH         Execute required function
          BRA.S   ST1            And issue prompt again
```

8.5 単純なコマンド・ルーチン

モニタの残りの部分は，コマンド探索テーブルを通じて呼び出される，多数のサブルーチンから構成されています．あらゆる場合において，これらのルーチンは，動作を引き起こした文字が，D0に入った状態で開始されます．そして，これらのルーチンは，A6(RAMワーク・エリアの基底に対するポインタが入っていると見なされる)を除いて，いずれかのレジスタを破壊する可能性があります．ルーチンはすべて，ユーザー・プログラムの実行に関係するもの(Gおよび

T)を除いて，呼出し元へ戻ります．ユーザー・プログラムを実行した場合，トラップ(TRAP #15)によってモニタへ戻ります．

最初のルーチンはデフォルト・ルーチンであり，未知のコマンドが入力された場合に呼び出されます．このルーチンは，単にメッセージを表示し，打ち込まれた入力行を終わりまで読み飛ばすため，標準ルーチンに入ります．

```
COMERR  LEA.L   MESS2,A0        Point to message
        BSR     WRITES          Print it
```

この次のサブルーチンは，ほとんどの他のコマンド・サブルーチンの終わりに呼び出されます．このサブルーチンは，入力行のコマンドのあとに続くものを単に読み取り，そして無視します．D0には，入力から最後に読み取られた文字が入っていなければなりませんが，この場合はキャリッジ・リターンを入力して終了します．さらにこのルーチンは，コンソールで打ち込まれたリターンに対する応答としても，呼び出されます．

```
SKIPNL  CMPI.B  #CR,D0          Check if d0 is CR
        BEQ.S   SKIPNL1         If so then exit
        BSR     RDCH            Otherwise ignore chars
        BRA.S   SKIPNL          Until it is one
SKIPNL1 RTS                     And return
```

次の一連のルーチンは，エラーが検出されたときに呼び出されます．最初のルーチンは，数が必要なのに見つからない場合に使用されます．このルーチンはSKIPNLを呼び出すので，レジスタD0には，行から取られた最後の文字が入っていなければなりません．

```
NUMERR  LEA.L   MESS3,A0        Point to message
        BSR     WRITES          Print it
        BRA.S   SKIPNL          And skip line
```

次のルーチンもほとんど同じで，無効なメモリ変更コマンドが検出された場合に呼び出されます．

```
MEMERR  LEA.L   MESS4,A0        Point to message
        BSR     WRITES          Print it
        BRA.S   SKIPNL          And Skip line
```

8.6 レジスタの表示と更新

次の各ルーチンは，ユーザー・レジスタの内容を表示し，変更するコマンドを取り扱います．最初は REGS で，R コマンドが入力されたあとに呼び出されます．これは単に，すべてのユーザー・レジスタの内容を表示するだけです．実際には，2つのエントリ・ポイントがあります．REGS は R コマンドが与えられた場合に使用され，REGX はトレースまたはブレーク・ポイント例外処理の後で，レジスタを表示する場合に使用されます．唯一の相違点は，REGS のエントリ・ポイントでは，残りのコマンド行(もしあれば)がスキップされる点です．

最初の数行は WRHEX4と WRHEX2をそれぞれ使って，ユーザー・プログラム・カウンタおよびステータス・レジスタを表示します．レジスタ A6は，RAM 作業領域の基底を参照するものと見なされる点に注意してください．

```
REGS    BSR     SKIPNL          Skip rest of line
REGX    MOVE.B  #'P',D0         Register letter into D0
        BSR     WRCH            Write it out
        MOVE.B  #'C',D0         And next letter
        BSR     WRCH            Write that
        BSR     BLANK           And a space
        MOVE.L  RDUMPPC(A6),D0  Obtain user PC
        BSR     WRHEX4          Write it out
        BSR     BLANK           Space
        MOVE.B  #'S',D0         Register letter
        BSR     WRCH            Write out
        MOVE.B  #'R',D0         And the next
        BSR     WRCH            Write that
        BSR     BLANK           And a aspace
        MOVE.W  RDUMPSR(A6),D0  Obtain user SR
        BSR     WRHEX2          Write out 2 bytes
        BSR     NEWLINE         And a newline
```

次の各行では，データ・レジスタとアドレス・レジスタを表示します．異なる２つのタイプの表示における類似性のために，REG1は，D1に文字〝D″が入り，A3がデータ・レジスタ保存領域の先頭を指した状態で，１回呼び出されるサブルーチンです．REG1が戻ったときには，A3は，アドレス・レジスタ保存領域の先頭を指した状態です．レジスタの文字は〝A″に更新され，REG1は，サブルーチンとして呼び出される形でなく，単に飛び込む形で再び使用されます．

```
          LEA.L    RDUMPD(A6),A3 Point to data registers
          MOVE.B   #'D',D1       Register letter into D1
          BSR.S    REG1          Display register set
          MOVE.B   #'A',D1       Register letter
 * .. and drop through
```

REG1セクションのコードでは，A3によって示されるメモリ番地に保存されている８個のユーザー・レジスタの値を表示します．D1に入っている文字を使ってレジスタを識別します．レジスタ番号を書き出すためにWRHEX0を，その値を書きだすためにWRHEX4を呼び出します．1行の中に適度なスペースを置いて４個のレジスタを書きだせるようにするため，必要に応じてサブルーチンBLANKおよびNEWLINEが呼び出されます．アドレス・レジスタA3は，レジスタ値が表示されると，レジスタ保存領域を指しつつインクリメントされます．したがって，タスクの完了時には，領域の終わりの直後の位置を指した状態になります．

```
REG1      MOVEQ    #0,D2         D2 is register number
REG2      MOVE.B   D1,D0         Extract register letter
          BSR      WRCH          Write register letter
          MOVE.B   D2,D0         Register number
          BSR      WRHEX0        Write nibble of that
          BSR      BLANK         Write out space
          ADDQ.B   #1,D2         Update register number
          MOVE.L   (A3)+,D0      Extract register value
          BSR      WRHEX4        Write it out
          CMP.B    #8,D2         All done yet?
          BEQ.S    REG3          All over
          BSR      BLANK         Print another space
          CMP.B    #4,D2         Register 4 next?
          BNE.S    REG2          No, so print once more
          BSR      NEWLINE       Newline before register 4
          BRA.S    REG2          And print next line
REG3      BRA      NEWLINE       Final newline and exit
```

特定のユーザー・レジスタの値を変更または確認するため，4つのコマンドが用意されています．AおよびDコマンドのあとにはレジスタ番号を指定します．レジスタ名のあとに何も値を指定しない場合は，単に値が表示されます．指定した場合は，その16進数値が読み込まれ，該当のレジスタがこの新しい値にセットされます．

DおよびAコマンドに対するルーチンは，レジスタA3がデータまたはアドレス・レジスタ保存領域の先頭を指すよう設定されると，共通のコードを使用するようになります．

```
SETD    LEA.L   RDUMPD(A6),A3  Set A3 to data reg store
        BRA.S   SETR           Jump to common code
*
* Address register
*
SETA    LEA.L   RDUMPA(A6),A3  Set A3 to addr reg store
```

この共通のコード・セクションでは，まずDまたはAのあとに指定されたレジスタ番号を読み込みます．READHEXを呼び出し，戻された値が有効であるかどうかをチェックします．有効でない場合，あらゆるレジスタ更新コマンドに対する共通のエラー・エグジットであるSETREへジャンプします．この値が正しい場合，4倍されてレジスタ記憶域に対するバイト・オフセットが計算されます．

```
SETR    BSR     READHEX
        BNE.S   SETRE          Register number expected
        TST.L   D1             Check bounds
        BLT.S   SETRE          Error
        CMP.L   #7,D1          Upper bound
        BGT.S   SETRE          Error
        MOVE.W  D1,D3          Save in D3
        ASL.W   #2,D3          Multiply by four
```

次のコードは，ユーザーのプログラム・カウンタを検査または変更するPコマンドによって，やはり共有されます．まず，入力から読み込まれた最後の文字が，リターンであるかどうかをチェックします．文字はREADHEXから，レジスタD0に戻されます．あとに続く値がない場合は，現在の値がラベルSETR2

のルーチンで表示されます.

行がリターンだけで終了しない場合は,READHEXのエントリ・ポイントREADHを使って指定された値が読み込まれます.これによって,D0にどんな文字が入っている場合でも,16進数値を読む際にその文字が考慮に入れられることになります.ここでエラーが検出されると,NUMERRにある標準コードにジャンプが行われます.エラーが検出されない場合は,その値が,正しいスロット(レジスタ番号と,データまたはアドレス・レジスタ保存領域のいずれかを指すベース・レジスタA3から計算されるオフセットとして示される)に挿入されます.最後にルーチンは,行に残った不要のテキストをスキップするSKIPNLを経て戻ります.

```
SETR1    CMP.B    #CR,D0        See if any value given
         BEQ.S    SETR2         No, so print value
         BSR      READH         Get value, last char in D0
         BNE      NUMERR        Hex number expected
         MOVE.L   D1,0(A3,D3.W)  Insert value in correct
*                               slot
         BRA      SKIPNL        Skip rest of line & return
```

新しい値がまったく指定されない場合は,ラベルSETR2に進みます.この場合,レジスタ保存領域の中の正しいスロットから現在の値が取り出され,表示されます.

```
SETR2    MOVE.L   0(A3,D3.W),D0 Extract register value
         BSR      WRHEX4        Print it out
         BRA      NEWLINE       Finish with NL
```

ユーザーのプログラム・カウンタに変更または確認するため,Pコマンドを使用した場合,サブルーチンSETPが呼び出されます.このプログラム・カウンタの値は,GまたはTコマンドのあとに値を指定することによっても更新することができます.SETPは単に,プログラム・カウンタ保存領域に対するポインタをA3にロードし,レジスタ番号に対するオフセットを0にセットします.これによって,ラベルSETR1にある共有コードに入ったとき,確実に正しい番地が参照されるようになります.ルーチンへジャンプする前に,入力からD0へ次の文字を読み込みます.この理由は前の場合においてREADHEXが呼び出され

るとき，最後に読み込まれた文字がD0に入るようセットされるためです．

```
SETP    LEA.L   RDUMPPC(A6),A3 Point to PC store
        CLR.W   D3             Offset zero
        BSR     RDCH           Get next character
        BRA.S   SETR1          Jump to shared code
```

Sコマンドは，ユーザーのステータス・レジスタを表示または変更するために使用されます．ステータス・レジスタはワード・サイズのオブジェクトなので，関連するサブルーチンは，今までに示したルーチンのように同一のコードを共用することはできません．また，ここでは，ステータス・レジスタの値をチェックしていません．

もしユーザーがトレースビットをセットした場合は，そのユーザーのコードがトレースされます．プログラムの開始時には，スーパーバイザビットがセットされていてはなりませんが，この段階ではチェックは何も行われません．

```
SETS    BSR     RDCH           Read next character
        CMP.B   #CR,D0         Check if new value given
        BEQ.S   SETS1          No, print current value
        BSR     READH          Get value, last char in D0
        BNE     NUMERR         Error in value
        MOVE.W  D1,RDUMPSR(A6) Update saved copy of SR
        BRA     SKIPNL         Return
SETS1   MOVE.W  RDUMPSR(A6),D0 Extract current value
        BSR     WRHEX2         Print it out
        BRA     NEWLINE        Print newline & return
```

レジスタ変更セクションの最後の部分は，単に無効なレジスタ番号が指定されたときにメッセージを表示します．

```
SETRE   LEA.L   MESS6,A0       Message
        BSR     WRITES         Print it out
        BRA     SKIPNL         Skip rest of line & return
```

8.7 ユーザー・プログラムの実行

本節では，ユーザー・プログラムへ入るためのT,GおよびCコマンドを実現しています．エントリ・ポイントTGOはTコマンド用に，GOはGコマンド用に使用されます．エントリ・ポイントTGOは，単に保存されているユーザーのステータス・レジスタの，トレースビットをセットします．エントリ・ポイントCONTは，Cコマンド用に使用され，ブレーク・ポイントからユーザー・プログラムの実行を継続します．

最初の段階は，ユーザー・プログラムに対するエントリ・ポイントが与えられているかどうかを調べることです．値がまったく与えられていない場合は，ラベルGO1にジャンプが行われます．与えられている場合は，D0に現在の文字がすでに入っているので，READHEXに対するエントリ・ポイントREADHを使って，ユーザー・プログラムのエントリ・ポイントが読み込まれます．正しいフォーマットならば保存されているユーザー・プログラム・カウンタの値が，新しい値に更新されます．

```
* Trace mode requested
TGO      BSET    #TBIT,RDUMPSR(A6) Set trace bit in
*                                  saved SR
* Normal mode requested
GO       BSR     RDCH              Get next character
         CMP.B   #CR,D0            Check for simple case
         BEQ.S   GO1               Start program running
         BSR     READH             Read entry point, D0 has
*                                  last character read
         BNE     NUMERR            Error in number
         MOVE.L  D1,RDUMPPC(A6)    Update saved PC
```

ラベルGO1で，ユーザー・プログラムが走り出します．このコードは，プログラムがトレース例外処理のために中断させられ，次の命令が再びトレースされるときに，例外処理手続きから入ることもできます．

最初の仕事は，ユーザーのステータス・レジスタのコピーにおいて，スーパーバイザビットがセットされていないことをチェックすることです．もしセットされている場合，プログラムは走りません．

```
GO1     BTST    #SBIT,RDUMPSR(A6) Check supervisor bit
*                                  not set
        BNE.S   GOERR             Error if so
        CLR.B   BFLG(A6)          Clear breakpoint flag
```

次の段階は，ユーザーのコードにブレーク・ポイントを挿入することです．ブレーク・ポイントは，プログラムが走り出す直前に挿入されるので，ユーザーが自分のコードを確認する場合，コードには変化ありません．ブレーク・ポイントが入るアドレスは，テーブル**BRKP**に収容されています．このテーブルには，1項目あたり6バイトが入っています．最初の4バイトには，ブレーク・ポイントのアドレスが入っており，このブレーク・ポイントが使用されない場合は，0が入ります．最後の2バイトには，2バイト命令**TRAP #14**によって置き換えられる，元のコードが入ります．ブレーク・ポイントのアドレスが0である場合は，それを検知したいのでアドレスをD1にロードします(**MOVEA**はコンディション・コードを変化させません)．この値をあとでアドレスとして使用したいので，A2を0にクリアします．すなわち，**0(A2,D1.L)**という文により，ユーザーはD1を実質的にはアドレス・レジスタとして使用できることになります．

ブレーク・ポイントを置いたメモリ番地に常駐するコードを，実際に実行することに関しては，もっと複雑な問題があります．ブレーク・ポイントに到達したら，2,3の命令についてトレースすることが非常に一般的です．この場合，ブレーク・ポイント・トラップを挿入するのではなく，実際にその命令を実行することが望ましいと言えます．この理由から，ブレーク・ポイントが，ユーザーのプログラム・カウンタによって指定されるアドレスと一致するかどうかもチェックします．このような場合，この時点ではブレーク・ポイントを挿入しません．実際には命令のコピーを使ってこれを実現します．というのは，例外処理手続きですべてのブレーク・ポイントを取り除く時点で，新たな例外的状況を引き起こすことなく，正しいコードに戻してやれるからです．

ブレーク・ポイントのところから継続するためには，特殊なコマンド**C**を使用しなければなりません．これは，ブレーク・ポイントが，現在のプログラム・カウンタ・アドレスにはセットされないという事実を利用しています．まず，トレースビットをセットし，次にラベル**CGO**にジャンプします．したがって，ブレーク・ポイント・アドレスにある命令を実行し，トレース例外処理のため

に，再びモニタに入ります．BFLGという特殊なフラグ(Cコマンドを与えた時点で0以外の値にセットされる)をセットします．例外処理手続きはこのフラグをチェックし，もしこれが0でなければ，単に標準Gルーチンを使ってプログラムを再開します．これによって，ブレーク・ポイントは確実に正しい位置に置き換えられ，必要ならば再実行することができます．すでにフラグBFLGを0にセットしているので，TまたはGが指定されていれば，この特殊な処理は行われません．

```
CGO     MOVEA.L RDUMPPC(A6),A4 Extract user PC
        LEA.L   BRKP(A6),A1   Point to breakpoint space
        MOVEQ   #9,D0         Counter
        SUBA.L  A2,A2         Zero A2
GO2     MOVE.L  (A1)+,D1      Breakpoint address
        BEQ.S   GO3           Zero address so no
*                             breakpoint
        MOVE.W  0(A2,D1.L),(A1) Save original
*                             instruction
        CMPA.L  D1,A4         Check if breakpoint at
*                             user PC
        BEQ.S   GO3           Do not insert breakpoint
*                             if so
        MOVE.W  #BRKTRP,0(A2,D1.L) Replace with
*                             breakpoint trap
GO3     ADDQ.L  #2,A1         Increment A1
        DBRA    D0,GO2        Try next breakpoint
```

これでユーザー・プログラムを走らせる準備が整いました．ユーザー・スタック・ポインタの保存されたコピーをレジスタA0に，次にユーザー・スタック・ポインタUSPに取り込みます．この処理が必要な理由は，USPに移動できるのはアドレス・レジスタだけだからです．

次の段階は，ユーザー・プログラム・カウンタおよびステータス・レジスタを取り出し，それらをシステム・スタックに保存して，後のRTE命令で使用できるようにすることです．次にMOVEMを使って，すべてのユーザー・レジスタを保存領域から再ロードし，次にRTEを実行してステータス・レジスタおよびプログラム・カウンタを再設定します．ステータス・レジスタは，スーパーバイザビットをクリアしてあるので，ユーザー・プログラムの実行は，ユーザーモードで行われます．プログラムがモニタへ制御を戻すのは，例外処理が発生した場合だけであり，ユーザー・プログラムが終了したことを示す方法とし

て，TRAP #15を確保してあります．

```
        MOVE.L  RDUMPSP(A6),A0 Extract user stack
*                              pointer
        MOVE.L  A0,USP         And set it up
        MOVE.L  RDUMPPC(A6),-(SP) Stack user PC
        MOVE.W  RDUMPSR(A6),-(SP) Stack user SR
        MOVEM.L RDUMPD(A6),D0-D7/A0-A6 Set up user's
*                              registers
        RTE                    Hold tight
```

C コマンドは，ブレーク・ポイントの後で継続するために使用します．スーパーバイザビットがセットされているのかどうかについての標準テストを行い，トレースビットをオンにし，コードはラベル CGO に分岐して，ユーザー・プログラムを開始します．ユーザー・プログラム・カウンタはブレーク・ポイント・アドレスと等しいので，この時点では，特定のブレーク・ポイントは挿入されません．また，条件 TRUE の Scc 命令を使って，フラグ BFLG を 0 以外の値にセットします．これは，例外処理手続きにおいて，適切なトレース例外処理と，ブレーク・ポイントの置かれていたコードの実行後に発生したトレース例外処理とを，区別するために使用されます．後者の場合，単にブレーク・ポイントをコードで置き換えて実行を続けます．これまでに示した手法により，ユーザーから見えるブレーク・ポイントが，常に正しく働くことが保証されます．

```
CONT    BSR     SKIPNL          Ignore any input
        BTST    #SBIT,RDUMPSR(A6) Check not supervisor
*                               bit
        BNE.S   GOERR           Error if so
        BTST    #TBIT,RDUMPSR(A6) Set trace bit
        ST      BFLG(A6)        Set marker flag to $FF
        BRA.S   CGO             Enter user program
```

最後に，保存されたステータス・レジスタ内でスーパーバイザビットがセットされていれば，GOERR に分岐が行われます．ここでは適切なエラー・メッセージを表示し，コマンド待ちに入ります．

```
GOERR   LEA.L   MESS7,A0        Load ptr to message
        BSR     WRITES          Write it out
        BRA     SKIPNL          Skip line & return
```

8.8 メモリの確認・更新ルーチン

次の一連のルーチンは，メモリを調べ変更するために使用します．Mという文字を打ち込んでこのコードに入ります．そのときに指定した番地が〝オープン〟され，そこに記憶されている値が表示されます．

次に，後続するメモリ変更コマンドが読み込まれ，それによってオープンされた番地の値が変更されたり，他の番地がオープンされたり，あるいは通常のコマンドモードに戻ったりします．

メモリ番地は，バイト，ワードまたはロングのオブジェクトとしてオープンすることができます．

初期的には，番地はバイトとしてオープンされます．オブジェクトのサイズはレジスタD2に入ります．番地がバイトとしてオープンされる場合はD2には1が入り，ワードの場合は2，ロングの場合は4がそれぞれ入ります．

現在のメモリ番地はレジスタA3に保持します．サブルーチンの最初の部分では，この番地を取り，そのアドレスを表示します．

```
MEM      BSR      READHEX      Read location
         BNE      NUMERR       Error in number
         MOVEA.L  D1,A3        Move address into A3
         MOVEQ    #1,D2        Set up as byte value
*                              initially
MEM1     MOVE.L   A3,D0        Move location into D0
         BSR      WRHEX4       And write it out
         BSR      BLANK        Write a space
```

次の段階ではD2に入っているサイズ指定を調べて，バイト，ワードまたはロング値を表示します．この処理には適切なサイズの値を取り出し，WRHEX1，WRHEX2，またはWRHEX4を使って，それを表示する処理が含まれます．

```
          CMP.B    #2,D2          Check size required
          BLT.S    MEMB           < 2 .. byte
          BEQ.S    MEMW           = 2 .. word
          MOVE.L   (A3),D0        Extract long data
          BSR      WRHEX4         Write out information
          BRA.S    MEMQ
MEMW      MOVE.W   (A3),D0        Extract word data
          BSR      WRHEX2         And write out
          BRA.S    MEMQ
MEMB      MOVE.B   (A3),D0        Extract byte
          BSR      WRHEX1         And write out
MEMQ      MOVE.B   #'?',D0        Question mark
          BSR      WRCH           Write that out
```

次の段階は新しい値を読み込むことです．エラーが発生したら，単純にエラー・メッセージを表示するだけではなく，有効なメモリ変更コマンドが与えられたかどうかを調べます．

このときD0には読み込まれた最後の文字が入ります．

```
          BSR      READHEX        Attempt to read new value
          BNE.S    MEM2           If not a number try other
*                                 command
          BSR      SKIPNL         Skip rest of line
```

有効な数が与えられた場合は，メモリ番地を更新しなければなりません．この処理を終えたら，同じメモリ番地を再び読む必要はありません．というのは，ACIA などの，I/O チップ内のレジスタであるメモリ番地に値を入れようとする場合に，問題が生じがちだからです．

NMEM という，次のメモリ番地に移動するサブルーチンがありますが，これを呼び出さなければなりません．それにはこのルーチンに BSR で分岐して，次に 3 つの異なるケースで MEM1に分岐するのではなく，まず，MEM1のアドレスを PEA 命令でスタックに置きます．次にルーチン NMEM に分岐し，NMEM の最後で RTS が実行されると MEM1に戻ります．

```
          PEA.L     MEM1          Push MEM1 so we will
*                                 return to it
          CMP.B     #2,D2         Check size again
          BLT.S     MEMBW         Byte
          BEQ.S     MEMWW         Word
          MOVE.L    D1,(A3)       Update long value
          BRA.S     NMEM          Display next value,
*                                 return to MEM1
MEMWW     MOVE.W    D1,(A3)       Update word value
          BRA.S     NMEM          Display next value,
*                                 return to MEM1
MEMBW     MOVE.B    D1,(A3)       Update byte value
          BRA.S     NMEM          And display next and
*                                 return to MEM1
```

無効な数が読み込まれた場合，違反した文字は，D0に入っています．同じルーチン **SEARCH** を使って，取るべき正しい動作を識別します．このとき渡されるのは，テーブル **MEMTAB** へのポインタです．これは，主実行ループで使用された **COMTAB** と同じ形式ですが，メモリ変更コマンドに対するアドレス・オフセットと文字が入っています．メモリ変更サブコマンドから戻った時点では，まだメモリ変更コマンドの環境に入ったままでいます．例外は，エグジット・コマンド(.)が与えられた場合で，もしそうならば **M** コマンドは終了します．さらに，いずれかのメモリ・コマンドが終了した後で，**SKIPNL** への呼び出しを通じて，入力行の残りの部分を読み飛ばします．

```
MEM2      CMPI.B    #'.',D0       Check for end command
          BEQ.S     MEM3          Exit if so
          LEA.L     MEMTAB,A0     Get memory change response
*                                 table
          BSR       SEARCH        Call suitable routine
          BSR       SKIPNL        Skip rest of line
          BRA.S     MEM1          Display again
MEM3      BRA       SKIPNL        Skip rest of line & return
```

以下の各ルーチンは，テーブル **MEMTAB** を通じて呼び出され，各種のメモリ・コマンドを実現しています．最初のルーチンは単に次のメモリ番地へ移動し，リターンが打ち込まれたときに呼び出されます．また，番地を更新した後にも呼び出されます．アドレスのサイズはD2で，着目している番地はA3に入っています．

```
NMEM     ADDA.L  D2,A3          Onto next location
         RTS
```

これも非常に類似しており，前の番地へ移動します．

```
PMEM     SUBA.L  D2,A3          Back to previous location
         RTS
```

次の各ルーチンはアクセスされるメモリのサイズを変更します．バイト・サイズの値の場合，単にD2に入っているサイズを変更することを意味します．

```
SETB     MOVEQ   #1,D2          Update D2
         RTS
```

番地がワードまたはロング値としてオープンされる場合，アドレスは，偶数でなければなりません．このチェックを行うために，サブルーチン CHKEVEN が呼び出されます．このサブルーチンは，チェックが成功した場合にのみ戻ります．それ以外の場合は，SETW または SETL を呼び出したルーチンにジャンプして戻ります．

```
* Set to word value.
SETW     BSR.S   CHKEVEN        Check even, error if not
         MOVEQ   #2,D2          Update D2
         RTS
* Set to long value.
SETL     BSR.S   CHKEVEN        Check if even
         MOVEQ   #4,D2          Update D2
         RTS
```

このルーチンは，A3に偶数アドレスが入っているかどうかをチェックします．このチェックが失敗すると，通常の戻り番地は無視され，そのルーチンを呼び出した，呼び出し元のルーチンへ戻ります．この場合，そのルーチンは常に MEM です．

```
CHKEVEN MOVE.L  A3,D0           Place A3 in D0
        BTST    #0,D0           Check bottom bit
        BEQ.S   CHK1            Zero so value is even
        LEA.L   MESS5,A0        Point to message
        BSR     WRITES          Print it
        LEA.L   4(SP),SP        Ignore this return address
CHK1    RTS                     Return or error return to
*                               MEM
```

8.9 ブレーク・ポイント

次のルーチンは，ブレーク・ポイントのセット，クリアおよび表示を取り扱います．すでに GO ルーチンのところで，ブレーク・ポイントのテーブルが管理されていることを説明しました．各ブレーク・ポイントについてアドレスと，オペコード(ブレーク・ポイントが挿入される際，置き換えられる)のために，6バイトが使用されています．

このセクションでは，ブレーク・ポイント・アドレスの処理だけを行います．B コマンドは単独で，現在のブレーク・ポイントを表示します．未使用のブレーク・ポイントは，アドレスが0にセットされています．そのため，アドレスがチェックされ，ブレーク・ポイントがセットされていれば，ブレーク・ポイント番号とアドレスが表示されます．

```
BRK     LEA     BRKP(A6),A1     Point to breakpoint table
        BSR     RDCH            Read next character
        CMP.B   #CR,D0          Check for simple B command
        BNE.S   BRK1            No, more complex
* Display current breakpoints
        MOVEQ   #0,D1           Counter
BRK0    TST.L   (A1)            Check if set
        BEQ.S   BRK01           No, not set
        MOVE.B  D1,D0           Breakpoint number into D0
        BSR     WRHEX0          Print breakpoint number
        BSR     BLANK           Print space
        MOVE.L  (A1),D0         Extract breakpoint location
        BSR     WRHEX4          Print address
        BSR     NEWLINE         Print newline
BRK01   ADDQ.L  #6,A1           Increment pointer
        ADDQ.B  #1,D1           Increment offset
        CMP.B   #9,D1           Check if done
        BLE.S   BRK0            Loop until done
        RTS                     Return
```

この場合，Bコマンドの後に，ブレーク・ポイント番号があるものと見なされます．前にRDCHを呼び出しているので，レジスタD0には，この後の文字が入ります．そのため，READHEXに対するREADHエントリ・ポイントを使ってブレーク・ポイント番号を得ます．

```
BRK1    BSR     READH           Read hex number, character
*                               in D0
        BNE.S   BRKE            Error in that
        TST.L   D1              Check within bounds
        BLT.S   BRKE            Too small
        CMP.L   #9,D1           Check other bound
        BGT.S   BRKE            Too big
```

ブレーク・ポイント番号が正しいことを確認できたら，正しいオフセットを求めなければなりません．1項目につき6バイトを使っているので，オフセットを求めるために，MULS命令を使用します．幸いなことに，許されるブレーク・ポイントは10個だけなので，MULSの引数のサイズに関する制約(1ワードに収まらなければならない)は，この場合影響ありません．次に，ブレーク・ポイント番号の後に，何かの値が与えられていないか調べます．何も与えられていない場合，ブレーク・ポイントをクリアし，与えられている場合は，新しいブレーク・ポイントをセットします．

```
        MULS    #6,D1           Offset in table
        ADDA.L  D1,A1           Point to slot
        CMP.B   #CR,D0          Any position given?
        BNE.S   BRK2
```

この場合A1によって示されているブレーク・ポイントをクリアします．

```
        CLR.L   (A1)            Clear breakpoint
        RTS                     And return
```

この段階では，指定された値を読み込み，ブレーク・ポイント・テーブルを更新しなければなりません．

```
BRK2    BSR     READH           Get position of breakpoint
        BNE     NUMERR          Error in that
        MOVE.L  D1,(A1)         Place address in slot
        BRA     SKIPNL          Skip rest and return
```

残る作業は，ブレーク・ポイント番号が無効である場合に，メッセージを表示することです．

```
BRKE     LEA.L    MESS8,A0       Point to message
         BSR      WRITES         Write message
         BRA      SKIPNL         Skip line & return
```

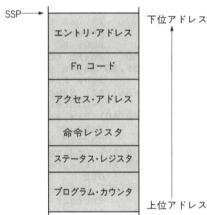
8.10 例外処理手続き

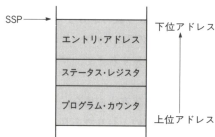
このコード・セクションでは，発生の可能性がある例外処理，割込み，トラップを処理します．標準的な動作は，必要に応じてユーザー・レジスタを保存したあと適切なメッセージを表示することです．

以下の各ラベルは，初期設定コードによって正しい例外処理ベクタに組み込まれているアドレスを定義するものです．各ラベルの箇所で，ショート型式ののBSR命令によって，起こり得る2つの異なるタイプの例外処理を扱うコードへ進みます．BSR を使用しているので，どの例外処理が発生したかを判定するためのインデックスとして，スタックに保存されている戻り番地を使用することができます．

```
* Exceptions
B_EXCPT BSR.S    EXCP1          Bus error
A_EXCPT BSR.S    EXCP1          Address error
I_EXCPT BSR.S    EXCP2          Illegal instruction
D_EXCPT BSR.S    EXCP2          Divide by zero
C_EXCPT BSR.S    EXCP2          CHK
O_EXCPT BSR.S    EXCP2          TRAPV
P_EXCPT BSR.S    EXCP2          Privilege
T_EXCPT BSR.S    EXCP2          Trace
X_EXCPT BSR.S    EXCP2          L1010
Y_EXCPT BSR.S    EXCP2          L1111
S_EXCPT BSR.S    EXCP2          Spurious interrupt
* Interrupts
INT     BSR.S    EXCP2          Unexpected interrupt
INT7    BSR.S    EXCP2          Level 7 interrupt
* Traps
TRP     BSR.S    EXCP2          Unexpected TRAP
TRP14   BSR.S    EXCP2          Breakpoint
TRP15   BSR.S    EXCP2          End of user program
```

EXCP1 では，より複雑なアドレスエラーまたはバスエラーを処理しなければなりません．68000では命令をプリフェッチするので，実際にエラーを起こした命令を示すプログラム・カウンタの値は，スタックに保存された値より小さくなる場合があります．スタックには多数の余分なワードの情報(命令レジスタも含む)が保存されています．プログラム・カウンタの位置にある命令が，命令レジスタに記憶されている命令と一致するまで，プログラム・カウンタを戻すことができます．

すべてのユーザー・レジスタを保存する場合，十分に注意しなければなりません．スタックのレイアウトは次のとおりであり，モニタに対するエントリ・ポイントのアドレスは，前の BSR によって直前に保存されたものです．

```
EXCP1     MOVEM.L D0/A0,-(SP)   Save some registers
          MOVE.L  22(SP),A0     Program counter
          MOVE.W  18(SP),D0     Instruction register
          CMP.W   -(A0),D0      Decrement PC and compare
          BEQ.S   EXCP10        Equal so ok
          CMP.W   -(A0),D0      Decrement again
          BEQ.S   EXCP10        Ok
          CMP.W   -(A0),D0      Decrement again
          BEQ.S   EXCP10        Ok
          CMP.W   -(A0),D0      Decrement again
          BEQ.S   EXCP10        Ok
          SUBQ.L  #2,A0         No so must be this one
EXCP10    MOVE.L  A0,22(SP)     Restore corrected PC
          MOVEM.L (SP)+,D0/A0   Restore saved registers
          MOVE.L  (SP),8(SP)    Overwrite with return addr
          ADDQ.L  #8,SP         Modify SP and drop through
```

これは，さらに単純なタイプの例外処理を表してます．システム・スタックは次のようになっています．

```
EXCP2   BTST     #SBIT,4(SP)  Test supervisor bit of
*                             saved SR
        BNE.S    EXCP3        If set then not user
*                             program running
        MOVE.L   A0,-(SP)     Save A0 temporarily
        MOVEA.L  RAMBASE,A0   Point to RAM area
        LEA.L    RDUMPD(A0),A0 Get a pointer to register
*                             save area
        MOVEM.L  D0-D7/A0-A6,(A0) Save all the user's
*                             registers
        MOVE.L   (SP)+,D_A0(A0) Fix saved value of A0
        MOVE.L   (SP)+,A1     Extract return address
*                             caused by BSR
        MOVE.W   (SP)+,D_SR(A0) Update user's SR
        BCLR     #TBIT,D_SR(A0) Ensure trace bit turned
*                             off
        MOVE.L   (SP)+,D_PC(A0) Update user's PC
        MOVE.L   USP,A2       Extract USP
        MOVE.L   A2,D_A7(A0)  And place that in A7 slot
```

これまでに，すべてのユーザー・レジスタを保存しましたが，次に，コードに挿入されたブレーク・ポイントを置き換えなければなりません．コードの元の値は，各6バイト領域の最後の2バイトに入っています．ユーザーのプログラム・カウンタがブレーク・ポイント・アドレスに等しかったために，ブレーク・ポイントが挿入されていなかったとしても，元の命令のコピーは，ブレーク・ポイント・テーブルに入ったままです．

```
        MOVEA.L RAMBASE,A6      Re-establish RAM pointer
        LEA.L   BRKP(A6),A3     Point to breakpoint save
*                               space
        MOVEQ   #9,D0           Counter
BRKL    MOVE.L  (A3)+,A4        Location of breakpoint
        MOVE.W  (A3)+,D1        Original code
        CMPA.L  #0,A4           Was breakpoint set?
        BEQ.S   BRKL1           No..
        MOVE.W  D1,(A4)         Replace original code
BRKL1   DBRA    D0,BRKL         Loop as required
        BRA.S   EXCP4           Now write message
```

モニタの実行中にエラー(幸いなことに，これはMコマンド実行中のバスエラーのみが起こり得る)が発生した場合には，ユーザー・レジスタを変更せず，通常どおりメッセージを表示します．

```
EXCP3   MOVE.L  (SP)+,A1        Extract return address
*                               stacked by BSR
```

これで，すべてのユーザー・レジスタを保存しました．A1には例外処理ベクタによってジャンプしたBSR命令の次の命令のアドレスが入っています．これを調整して，実際の命令を指すようにしなくてはなりません．

さらに，システム・スタックをリセットして，モニタに入ったときに与えられていた元の値にしなければなりません．この処理が終わったら，再び安全に割込みをオンにできます．というのは，予測されない割込みが多数あったとしても，他の割込みを処理しようとする前に，スタックを再び基底にリセットするからです．

```
EXCP4   SUBQ.L  #2,A1           Pointer to code we
*                               actually entered
        MOVE.L  I_RESET,SP      Reset system stack
        MOVE.W  #INTSON,SR      Interrupts on again
```

ここで，2つの特殊な状況について見てみましょう．それらは，トレース例外処理とブレーク・ポイントです．

```
        LEA.L    T_EXCPT,A0     Trace exception
        CMPA.L   A0,A1          Was it one?
        BEQ.S    EXCP5          Yes, handle it
        LEA.L    TRP14,A0       Breakpoint trap?
        CMPA.L   A0,A1          Was it this?
        BEQ.S    EXCP6          Handle it
```

この時点では，他のタイプの例外処理でした．A1の値に基づいてメッセージを書き出すサブルーチン WRABO を呼び出します．A1の値は，打切りを示すエントリ・ポイントを指したままであり，正しいメッセージを選択するために使用されます．この処理が終わったら，コマンド・ループの先頭へ分岐して，コマンド待ちに入ります．

```
        BSR.S    WRABO          Write suitable message
        BRA      ST1            And handle any more
*                               commands
```

トレース例外処理が発生した場合にこの部分にきます．トレース例外処理の原因としては2つあります．第一は，C コマンドによって発生した場合です．C コマンドでは，トレースビットをセットします．ブレーク・ポイントとして使われる TRAP #14命令で上書きされた命令は，通常に実行されたので，C コマンドはトレースビットをセットし，ブレーク・ポイントを置き換えます．この場合，フラグ BFLG は0以外の値になり，単にユーザー・プログラムに再び入ります．その後ブレーク・ポイントは，次の使用のために再び戻されます．

```
EXCP5   TST.B    BFLG(A6)       Test to see if C was last
*                               command
        BNE      GO1            Continue execution if so
```

これは，通常のトレース例外処理を扱います．まず，WRABO を再び使って，適切なメッセージを書き出します．次に REGSのエントリ・ポイントREGX を呼び出してレジスタを表示します．

トレースが次々と必要になることは非常によくあることなので，通常のコマンド処理を修正して，〝リターン〟を押すことが T を打ち込むことと同じになるようにします．これ以外のコマンドは，通常どおり処理されます．この処理を

行うために，A0がコマンド・テーブルを指すようにし，次に PEA を使って，主コマンド・ループのエントリ・ポイントをスタックに置きます．

モニタが特殊なモードにあることを示すプロンプトとして小文字〝t〞を書き出し，次にユーザーの応答を読み込みます．この応答が単なる〝リターン〞ではない場合，それを処理するために SEARCH サブルーチンに分岐します．ST1 のアドレスをスタックに置いているので，SEARCH によって呼び出されたサブルーチンが最終的に戻る場合，ここに戻るのではなく，ST1に戻ります．

読み込まれた文字がリターンである場合には，トレースビットをセットして，GO1 にジャンプすることにより，ユーザー・プログラムの実行を続けます．

```
        BSR.S   WRABO           Write trace message
        BSR     REGX            Print registers
        LEA.L   COMTAB,A0       Point to command table
        PEA.L   ST1             Push return address of
*                               command loop
        MOVE.B  #'t',D0         New prompt character
        BSR     WRCH            Write it out
        BSR     RDCH            Get next character
        CMP.B   #CR,D0          Return?
        BNE     SEARCH          No, do standard search for
*                               command
        BSET    #TBIT,RDUMPSR(A6) Set the trace bit in
*                               saved SR
        BRA     GO1             And continue execution
```

この時点で，ブレーク・ポイントの処理を行わなければなりません．ユーザーは，TRAP #14命令で置き換えたアドレスに，ブレーク・ポイントをセットします．TRAP 命令は１ワード長のみ(最も短い命令と同じ長さ)なので，これは常にうまくいきます．この TRAP #14によってここに到達したことになりますが，ブレーク・ポイントがセットされていなければあるはずの命令は，まだ実行していません．したがって，最初に行うべき処理は，プログラム・カウンタを２だけデクリメントすることです．

次に適切なメッセージを書き出し，REGX を呼び出して，レジスタの状態を表示します．次に後続のコマンドを読み込むために ST1へ分岐します．ユーザーがプログラムの続行を希望する場合，このトラップを発生させたブレーク・ポイントを挿入しません．というのは，このブレーク・ポイント・アドレスが，プログラム・カウンタに等しくなるからです．C または T コマンドを使用した

場合，命令が実行された直後に，モニタに制御が戻ります．そしてその後，ユーザー・プログラムが次に再開するとき，TRAP #14命令と置き換えます．

```
EXCP6   SUBQ.L  #2,RDUMPPC(A6) Back up user PC
        BSR.S   WRABO          Write breakpoint message
        BSR     REGX           Display registers
        BRA     ST1            Ask for another command
```

モニタの最後のサブルーチンは，レジスタA1に入っている値を使って，発生した例外処理に対応する，適切なメッセージを書き出します．

すでにA1を調整して，例外処理手続きに入ったラベルのアドレスを示すようにしてあります．起こり得る例外処理の最初のラベルのアドレスを，A0にロードして，2を減算します．個々のBSR.S命令が1ワードを占めるので，結果は例外のタイプに対応するワード・オフセットとなります．これは，テーブルABOTABに対するインデックスとして，あとで使用されます．ABOTABの各々の項目は，エラーを記述する文字列のテーブルの基底からのオフセットです．テーブルの基底に文字列のオフセットを加算して文字列のアドレスを求め，WRITESを使ってこれを書き出します．

```
WRABO   LEA.L   B_EXCPT,A0     Base of table
        SUBA.L  A0,A1          Now a word offset from
*                              zero
        LEA.L   ABOTAB,A2      Pointer to abort table
*                              base
        MOVE.L  A1,D0          Offset into D0
        MOVE.W  0(A2,D0.L),A0  Offset of string from
*                              table base
        ADDA.L  A2,A0          Add table base to point
*                              to string
        BRA     WRITES         Write it out and return
```

8.11 メッセージとテーブル

このあとは，モニタで使用するメッセージとテーブルを定義するだけになりました．まず，ユーザーのタイプミスの結果として発生するエラー・メッセージと，モニタのオープニング・メッセージです．

```
MESS1   DC.B    'MC68000 monitor V1.2',0
MESS2   DC.B    'Unknown command',0
MESS3   DC.B    'Hexadecimal number expected',0
MESS4   DC.B    'Invalid memory command',0
MESS5   DC.B    'Current address not even',0
MESS6   DC.B    'Invalid register number',0
MESS7   DC.B    'Supervisor bit set',0
MESS8   DC.B    'Invalid breakpoint number',0
```

次は，例外処理，予期しない割込み，およびトラップに関するメッセージです．

```
AB1     DC.B    'Bus error',0
AB2     DC.B    'Address error',0
AB3     DC.B    'Illegal instruction',0
AB4     DC.B    'Division by zero',0
AB5     DC.B    'CHK exception',0
AB6     DC.B    'TRAPV exception',0
AB7     DC.B    'Privilege violation',0
AB8     DC.B    'Trace...',0
AB9     DC.B    'Illegal instruction (1010)',0
AB10    DC.B    'Illegal instruction (1111)',0
AB11    DC.B    'Spurious interrupt',0
AB12    DC.B    'Unexpected interrupt',0
AB13    DC.B    'Level 7 interrupt',0
AB14    DC.B    'TRAP exception',0
AB15    DC.B    'Breakpoint',0
AB16    DC.B    'End of user program',0
```

これらのメッセージのアドレスは，次のテーブルに記憶します．位置の独立性を保つため，テーブルの基底からのオフセットとして記憶します．テーブル内での順序は，例外処理手続きに入るために使用されるラベルの順序に対応しています．

```
ABOTAB  DC.W    (AB1-ABOTAB)
        DC.W    (AB2-ABOTAB)
        DC.W    (AB3-ABOTAB)
        DC.W    (AB4-ABOTAB)
        DC.W    (AB5-ABOTAB)
        DC.W    (AB6-ABOTAB)
        DC.W    (AB7-ABOTAB)
        DC.W    (AB8-ABOTAB)
        DC.W    (AB9-ABOTAB)
        DC.W    (AB10-ABOTAB)
        DC.W    (AB11-ABOTAB)
        DC.W    (AB12-ABOTAB)
        DC.W    (AB13-ABOTAB)
        DC.W    (AB14-ABOTAB)
        DC.W    (AB15-ABOTAB)
        DC.W    (AB16-ABOTAB)
```

次の2つのテーブルは，SEARCH サブルーチンに対する正しい形式になっています．すなわち，各々の4バイト項目の最初のバイトは，テーブル内の最後の項目を除いて，0になっています．2番目のバイトには，文字が小文字形式で入っており，次の2バイトは，読み込まれた文字が，項目内の文字と一致している場合に，呼び出すべきサブルーチンを参照します．このサブルーチンは，テーブルの基底からのオフセットとして指定します．各テーブルの最後の項目は，最初のバイトが0以外の値にセットされており，指定されたルーチンが常に呼び出されます．

最初のテーブルは，通常のコマンドを処理します．

```
COMTAB  DC.W    CR              Just a return
        DC.W    (SKIPNL-COMTAB)
        DC.W    'm'             Memory change
        DC.W    (MEM-COMTAB)
        DC.W    'r'             Register dump
        DC.W    (REGS-COMTAB)
        DC.W    'd'             Alter data register
        DC.W    (SETD-COMTAB)
        DC.W    'a'             Alter address register
        DC.W    (SETA-COMTAB)
        DC.W    'p'             Alter PC
        DC.W    (SETP-COMTAB)
        DC.W    's'             Alter SR
        DC.W    (SETS-COMTAB)
        DC.W    'g'             Enter user program
        DC.W    (GO-COMTAB)
        DC.W    't'             Trace user program
        DC.W    (TGO-COMTAB)
        DC.W    'b'             Breakpoint
```

```
        DC.W    (BRK-COMTAB)
        DC.W    'c'           Continue after breakpoint
        DC.W    (CONT-COMTAB)
        DC.W    $FF00         Marker flag for end
        DC.W    (COMERR-COMTAB)
```

最後に，M コマンドの後に指示されるメモリ・サブコマンドをデコードするためのテーブルを指定します．

```
MEMTAB  DC.W    CR            Move to next location
        DC.W    (NMEM-MEMTAB)
        DC.W    '^'           Up arrow
        DC.W    (PMEM-MEMTAB) Previous memory location
        DC.W    '='           Equals
        DC.W    (SKIPNL-MEMTAB) Stay at same location
        DC.W    's'           Set to byte size
        DC.W    (SETB-MEMTAB)
        DC.W    'w'           Set to word size
        DC.W    (SETW-MEMTAB)
        DC.W    'l'           Set to long size
        DC.W    (SETL-MEMTAB)
        DC.W    $FF00         Marker flag for end
        DC.W    (MEMERR-MEMTAB) Memory change error
        END
```

付　　録

68000命令セット

命令の後のステータス・レジスタ内のコンディション・コード・フラグの状態は，次のように示されます．

−	影響されない
0	常にクリアされる
1	常にセットされる
A	デスティネーションがアドレス·レジスタの場合を除いて変化する
C	値に応じて変化する
P	値に応じて変化する可能性がある
U	不定

アドレスモードは次のように示されます．

An	任意のアドレス・レジスタ
Dn	任意のデータ・レジスタ
Rn	任意のレジスタ
(An)	アドレス・レジスタ間接
d(An)	ディスプレースメント付きアドレス・レジスタ間接
−(An)	プレデクリメント付きアドレス・レジスタ間接
(An)+	ポストインクリメント付きアドレス・レジスタ間接
<ea>	任意のアドレスモード
<aea>	可変アドレスモード
<cea>	制御アドレスモード
<dea>	データ・アドレスモード
<caea>	制御可変アドレスモード
<daea>	データ可変アドレスモード
<maea>	メモリ可変アドレスモード
<rl>	レジスタ・リスト
<imm>	イミディエイト・データ

次の表では，各種の実効アドレスを分類します．

モード	データ	メモリ	制御	可変
Dn	*			*
An				*
d16(PC)	*	*	*	
d8(PC, Ri)	*	*	*	
(An)	*	*	*	*
d16(An)	*	*	*	*
d8(An, Ri)	*	*	*	*
−(An)	*	*		*
(An)+	*	*		*
アブソリュート	*	*	*	*
#データ	*	*		

各命令のサイズは，バイトの場合は B，ワードの場合は W，ロングの場合は L で示します．命令の実行後のコンディション・コードの状態を次に示し，その後に，その命令に関する詳細な説明が掲載されているページ番号を示します．

最後に，各種の形式のアドレスモードの例を示します．同じ形式の命令が多数ある場合は，最初の命令だけを例で示します．異なる命令で異なる構文を使用する場合は，個々の使用可能なバリエーションを示します．

名前 ※：特権化命令	説　　明	サイズ	NZVCX	ページ
ABCD	10進数の加算	B	U P U C C	140
	モード：ABCD Dn, Dn			
	ABCD −(An), −(An)			
ADD	2進数の加算	B W L	C C C C C	124
ADDA		W L	− − − − −	125
ADDI		B W L	C C C C C	125
ADDQ		B W L	A A A A A	125
ADDX		B W L	C P C C C	126
	モード：ADD 〈ea〉, Dn			
	ADD Dn, 〈maea〉			
	ADDA 〈ea〉, An			
	ADDI ♯〈imm〉, 〈daea〉			
	ADDQ ♯〈imm〉, 〈aea〉			
	ADDQ Dn, Dn			
	ADDX −(An), −(An)			
AND	論理的AND	B W L	C C O O −	149
ANDI		B W L	C C O O −	150
ANDI to CCR		B	P P P P P	150
※ANDI to SR		W	P P P P P	150
	モード：AND 〈dea〉, Dn			
	AND Dn, 〈maea〉			
	ANDI ♯〈imm〉, 〈daea〉			
	ANDI ♯〈imm〉, CCR			
	ANDI ♯〈imm〉, SR			
ASL	左への算術シフト	B W L	C C C C C	152
ASR	右への算術シフト	B W L	C C C C C	152
	モード：ASL Dn, Dn			
	ASL ♯〈imm〉, Dn			
	ASL 〈maea〉			
BCC	条件付き分岐	B W	− − − − −	75
BRA	無条件分岐	B W	− − − − −	76
BSR	サブルーチンへの分岐	B W	− − − − −	100
	モード：BCC 〈ラベル〉			

名前 ※:特権化命令	説　　明	サイズ	NZVCX	ページ
BCHG	ビットテストおよび変更	B　L	－C－－－	155
BCLR	ビットテストおよびクリア	B　L	－C－－－	155
BSET	ビットテストおよびセット	B　L	－C－－－	155
BTST	ビットテスト	B　L	－C－－－	155
	モード：BCHG Dn, <daea>			
	BCHG #<imm>, <daea>			
	BTST Dn, <dea>			
	BTST #<imm>, <dea>			
CHK	チェックおよびTRAP	W	PUUU－	175
	モード：CHK <dea>, Dn			
CLR	ゼロにセット	BWL	0100－	79
	モード：CLR <daea>			
CMP	比較	BWL	CCCC－	72
CMPA		WL	CCCC－	73
CMPI		BWL	CCCC－	73
CMPM		BWL	CCCC－	74
	モード：CMP <ea>, Dn			
	CMPA <ea>, An			
	CMPI #<imm>, <daea>			
	CMPM (An)+, (An)+			
DBcc	デクリメント，テストおよび分岐	W	－－－－－	83
DBRA	デクリメントおよび分岐	W	－－－－－	84
	モード：DBcc Dn, <ラベル>			
DIVS	除算(符号付き)	W	CCCO－	135
DIVU	除算(符号なし)	W	CCCO－	135
	モード：DIVS <dea>, Dn			

名前 ※:特権化命令	説　　明	サイズ	NZVCX	ページ
EOR EORI EORI to CCR ※EORI to SR	論理的排他的OR モード：EOR Dn, <daea> EORI ＃<imm>, <daea> EORI ＃<imm>, CCR EORI ＃<imm>, SR	BWL BWL B W	CCOO－ CCOO－ PPPPP PPPPP	149 150 150 150
EXG	レジスタの交換 モード：EXG Rn, Rn	L	－－－－－	129
EXT	符号拡張 モード：EXT Dn	WL	CCOO－	129
JMP JSR	ジャンプ サブルーチンへのジャンプ モード：JMP <cea>	－ －	－－－－－ －－－－－	107 108
LEA	実効アドレスのロード モード：LEA <cea>, An	L	－－－－－	109
LINK	サブルーチンのリンク モード：LINK An, ＃<imm>	－	－－－－－	115
LSL LSR	左への論理シフト 右への論理シフト モード：LSL Dn, Dn LSL ＃<imm>, Dn LSL <maea>	BWL BWL	CCOCC CCOCC	151 152

名前 ※:特権化命令	説　明	サイズ	NZVCX	ページ
MOVE	データの移動	BWL	CCOO—	66
MOVEA		WL	—————	68
MOVEM	複数レジスタとメモリとの移動	WL	—————	96
MOVEP	周辺装置への移動	WL	—————	90
MOVEQ		L	CCOO—	81
MOVE to CCR		W	CCCCC	170
※MOVE to SR		W	CCCCC	170
MOVE from SR		W	—————	170
※MOVE USP		L	—————	168
	モード：MOVE <ea>, <daea>			
	MOVEA <ea>. An			
	MOVEM <rl>, —(An)			
	MOVEM <rl>, <caea>			
	MOVEM (An)+, <rl>			
	MOVEM <cea>, <rl>			
	MOVEP Dn, d(An)			
	MOVEP d(An), Dn			
	MOVEQ ＃<imm>, Dn			
	MOVE <dea>, CCR			
	MOVE <dea>, SR			
	MOVE SR, <daea>			
	MOVE USP, An			
	MOVE An, USP			
MULS	乗算(符号付き)	W	CCOO—	128
MULU	乗算(符号なし)	W	CCOO—	128
	モード：MULS <dea>, Dn			
NBCD	10進数の負数をとる	B	UPUCC	140
	モード：NBCD <daea>			
NEG	2進数の負数をとる	BWL	CCCCC	127
NEGX		BWL	CPCCC	127
	モード：NEG <daea>			

名前 ※:特権化命令	説　　明	サイズ	NZVCX	ページ
NOP NOT	無動作 モード：NOP 論理的否定 モード：NOT <daea>	－ BWL	－－－－－ CCOO－	38 148
OR ORI ORI to CCR ※ORI to SR	論理的OR モード：OR <dea>, Dn OR Dn, <maea> ORI ＃<imm>, <daea> ORI ＃<imm>, CCR ORI ＃<imm>, SR	BWL BWL B W	CCOO－ CCOO－ PPPPP PPPPP	148 150 150 150
PEA	実効アドレスのプッシュ モード：PEA <cea>	L	－－－－－	111
※RESET	リセット モード：RESET	－	－－－－－	176
ROL ROXL ROR ROXR	左へのローテイト 右へのローテイト モード：ROL Dn, Dn ROL ＃<imm>, Dn ROL <maea>	BWL BWL BWL BWL	CCOC－ CCOCC CCOC－ CCOCC	153 153 153 153
※RTE RTR RTS	例外処理からのリターン リターンおよびCCRの復元 サブルーチンからのリターン モード：RTE	－ － －	CCCCC CCCCC －－－－－	170 170 101
SBCD	10進数の減算 モード：SBCD Dn, Dn SBCD －(An), －(An)	B	UPUCC	140

名前 ※:特権化命令	説　　明	サイズ	NZVCX	ページ
Scc	条件によるセット モード：Scc 〈daea〉	B	―――――	82
※STOP	実行停止および待機 モード：STOP ＃〈imm〉	―	CCCCC	176

SUB
SUBA
SUBI
SUBQ
SUBX
	2進数の減算
	
	モード：SUB 〈ea〉, Dn
	SUB Dn, 〈maea〉
	SUBA 〈ea〉, An
	SUBI ＃〈imm〉, 〈daea〉
	SUBQ ＃〈imm〉, 〈aea〉
	SUBX Dn, Dn
	SUBX −(An), −(An)
		BWL
		WL
		BWL
		BWL
		BWL
			CCCCC
			―――――
			CCCCC
			CCCCC
			CPCCC
				126
				127
				127
				127
				127

SWAP	レジスタの上下のスワップ モード：SWAP Dn	W	CCOO―	129
TAS	テストビットおよびセット モード：TAS 〈daea〉	B	CCOO―	156
TRAP TRAPV	トラップ例外処理の発生 オーバーフローの場合にTRAP モード：TRAP ＃〈imm〉 TRAPV	― ―	――――― ―――――	174 175
TST	0との比較 モード：TST 〈daea〉	BWL	CCOO―	80
UNLK	サブルーチンのアンリンク モード：UNLK An	―	―――――	116

■条件テスト

以下のテストは，Bcc，DBcc，および Scc の各命令で，条件テストとして指定できるものです．その他に，DBcc，および Scc 命令で真および偽を示すために，T および F を使用することができます．ある種のアセンブラでは，DBF に対する追加的な表記，DBRA と，CC および CS に対するテスト，HS および LO の使用を認めています．

以下の表で，C は，条件が真となるために，C ステータス・フラグがセットされていなければならないことを示し，C'は，このフラグがクリアされていなければならないことを示します．個々の条件を，&(両方とも真でなければならないの意)または ¦ (いずれかが真であればよいの意)で結合することができます．

名前	条　件	テ　ス　ト
CC	キャリークリア	C'
CS	キャリーセット	C
EQ	等しい	Z
NE	等しくない	Z'
PL	正	N'
MI	負	N
VC	オーバーフロークリア	V'
VS	オーバーフローセット	V
HI	高い	C' & Z'
LS	低いか同じ	C ¦ Z
HS	高いか同じ	C'
LO	低い	C
GT	より大きい	(N&V&Z') ¦ (N' &V' &Z')
GE	より大きいか等しい	(N&V) ¦ (N' &V')
LE	より小さいか等しい	Z ¦ (N&V') ¦ (N' &V)
LT	より小さい	(N&V') ¦ (N' &V)

索　　引

ア

アセンブラ……38
アセンブラ・ディレクティブ……41,45
アセンブリ言語……38
アドレッシングモード……60
アドレス……22,42
アドレスエラー……179
アドレス・レジスタ……22
アドレス・レジスタ間接……54
アブソリュート……25
アブソリュート・アドレッシング……49
アブソリュート記号……47
アブソリュート形式……66
アブソリュート・コード……42
アブソリュート・ジャンプ……106
位置独立コード……25
イミディエイト・アドレスモード……66
イミディエイト・データ……58
インテル8086……16
インテルiAPX 286……20
インテルiAPX 432システム……21
インプリシット・アドレッシング……60
エグジット手続き……115
エントリ手続き……114
オブジェクト……94
オペランド……39
オペランド・ワード……23
オペレーション・ワード……23
オペレーティング・システム……30
重み付け因子……137
オリジン……42

カ

外部的な例外処理……164
加算命令……124
間接アドレッシング……54
偽(論理値)……148
機械語……28,38
減算命令……126
高級言語……28
交差積……132
コメント……40
コンディション・コード……82
コンディション・コード・フラグ……24
コンパイラ……28

サ

サイズ指示子……39,46
ザイログZ80……16
ザイログZ8000……18
サフィックス……39
サブルーチン……102
算術演算……124
算術演算子……46
資源……31
実効アドレス……49

実効アドレスのプッシュ……………………111
実効アドレスのロード………………………109
システムバイト………………………………24
自動ベクタ……………………………………172
ジャンプ………………………………………23
主記憶…………………………………22,23
10進算術演算…………………………………139
16ビットマシン………………………………14
順次再使用可能なコード……………………51
乗算命令………………………………………128
除算命令………………………………………135
条件付き分岐…………………………………70
シリアル・ライン……………………………86
真(論理値)……………………………………148
シングルチップ・コンピュータ……………14
スタック………………………………………94
スタックの先頭………………………………94
スタック・ポインタ…………………………94
ステータス・レジスタ………………………23
スーパーバイザ・スタック…………………169
スーパーバイザ・スタック・ポインタ…99
スーパーバイザビット………………………24
スーパーバイザモード………………30,167
スペアビット…………………………………22
絶対番地………………………………………112
セマフォ………………………………………156
ソース…………………………………………39

タ

ダイレクト・メモリ・アクセス…………32
ダブルバス・フォルト………………………181
ディスプレースメント付きレジスタ間接…55
デスティネーション…………………………39
データの移動…………………………………25
データ・レジスタ……………………………22
デマンドページ方式…………………………20
特権化…………………………………………176
トラップ………………………27,31,164,222
トレースビット………………………………24
トレースモード………………………………27
トレース例外処理……………………178,226

ナ

内部的な例外処理……………………………164
ニブル…………………………………………140
ニーモニック名………………………………38
ノンマスカラブル・インタラプト……172

ハ

排他的アクセス………………………………31
排他的OR……………………………………149
排他的な使用権………………………………173
倍長の除算ルーチン…………………………136
倍長の乗算ルーチン…………………………131
バイト(8ビット)…………………………22,39
バス調停回路…………………………………31
バスエラー……………………………………179
8ビットマシン………………………………14
ハードウェア・トラップ……………………27
ビットテスト…………………………………81
非同期通信インターフェース・アダプタ…86
ピュア・コード………………………………51
符号拡張………………………………………49
不正命令………………………………………174
フラグ…………………………………………31
ブレーク・ポイント…………………178,220
プレデクリメント付きレジスタ間接……57
プレデクリメントモード……………………57
プレフェッチ…………………………………98
プロセッサモード……………………………30
分岐……………………………………………23
分岐テーブル…………………………………201
ベクタ付き割込み……………………………31
ベクタ番号……………………………………166
補数演算………………………………………134
ポストインクリメント付きレジスタ間接…57
ポストインクリメントモード………………58
ポーリングモード……………………………87

マ

マイクロプロセッサ……13
マスキング……149
未実装命令……173
命令キャッシュ……34
メモリ……22
モニタ……30,86,188
戻り番地……100
モトローラ6809……16
モトローラ68000……19

ヤ

ユーザー・スタック・ポインタ……99
ユーザーバイト……24
ユーザーモード……30,167

ラ

ラブアウト……194
ラベル……40
リラティブ……25
リラティブ・アドレッシング……51
リエントラント……51
リセット例外処理……173
リード・モディファイ・ライト・サイクル……31,156
リロケーション情報……44,107
リロケータブル……43
リロケータブル記号……47
リロケータブル・シンボル……43
例外処理……164,181,222
例外処理ベクタ……164,165
レジスタ……22
レジスタ直接アドレッシング……49
68000……15,21
68008……33
68010……33
68020……34
ローテイト命令……153
ロングワード(32ビット)……22,39
論理演算……148
論理シフト命令……151

ワ

ワード(16ビット)……22,39
割込み……164,222
割込み処理……172
割込みハンドラ……31
割込みマスク……24,172

A

ABCD……140,143
ACIA……86
ADD……124
ADDA……125
ADDI……125
ADDQ……125
ADDX……126
AND……149
ANDI……150
ASL……152
ASR……152
A7(アドレス・レジスタ)……99

B

BCC……75
Bcc……75,82
BCD……140
BCHG……155
BCLR……155
BCS……75
BEQ……70,75
BF……76
BGE……75
BGT……76
BHI……76
BHS……76
BLE……76
BLO……76
BLS……76

BLT……75
BMI……75
BNE……70,75
BPL……75
BRA……76,100
BSET……155
BSR……76,100
BT……76
BTST……155
BVC……75
BVS……75

C

C(コンディション・コード・フラグ)……25
CHK……27,175
CMP……72
CMPA……73
CMPI……73
CMPM……74,144

D

DBcc……76,83,119
DBF……84
DBRA……84
DCディレクティブ……44
DIVS……135,175
DIVU……135,137,175
DMA……32
DRAM……14
DSディレクティブ……44

E

ENDディレクティブ……45
EOR……149
EORI……150
EQU……42,78
exceptions……164
exclusive access……173
EXG……129
EXT……129

F

FPU……20

J

JMP……43,106
JSR……108

L

LEA……109
LINK……29,115
LSL……151
LSR……152

M

MMU……20
MOSテクノロジー6502……15
MOVE……66,96
MOVEA……68
MOVEC……33
MOVEM……29,31,96
MOVEP……31,90
MOVEQ……59,81
MOVES……33
MULS……128
MULU……128

N

N(コンディション・コード・フラグ)……24
Nビットプロセッサ……14
NBCD……140,144
NEG……127
NEGX……127
nibble……140
NOT……148
NS32016……20

O

OR ……148
ORG ……42, 62
ORI ……150

P

PEA ……111

R

RAM ……14
RESET ……176
ROL ……153
ROM ……14
ROR ……153
RORG ……43, 63
ROXL ……153
ROXR ……153
RTE ……168, 170, 194
RTR ……170
RTS ……101

S

SBCD ……140, 144
scaled factor ……137
Scc ……76, 82
semaphore ……156
SP ……101
SSP ……99
STOP ……176
SUB ……126, 143
SUBA ……127
SUBI ……127
SUBQ ……127
SUBX ……127
SWAP ……129

T

TAS ……31, 156
TMS99000 ……18
TMS9900シリーズ ……17
TOS ……94
TRAP ……27, 31, 174
TRAPV ……27, 175
TST ……80

U

UNLK ……29, 116
USP ……99

V

V(コンディション・コード・フラグ) ……24
VBR ……33

X

X(コンディション・コード・フラグ) ……25

Z

Z(コンディション・コード・フラグ) ……24

記号

$ ……43
$15 ……87
……59
#値 ……58
* ……40, 45, 52
+ ……46
− ……46
*(乗算) ……46
/(除算) ……46
.B ……39
.L ……39
.W ……39
.S ……39, 72
_ ……41
' ' ……47

■著者略歴

Dr.Tim King

ケンブリッジ大学で，データベース管理システムの博士号を取得．

68000に関する，ローカルネットワークのデータベース・オペレーティング・システムを専門とし，1980年，バース大学でソフトウェア技術の講義を受けもつ．

1984年，68000のソフト開発を専門とするソフトウェアハウスの取締役に就任するとともに，バース大学の特別研究員としても活躍中である．

Dr.Brian Knight

ケンブリッジ大学コンピュータ研究所で博士号を取得．

ローカルネットワークにおける68000と他のマシン用のソフトウェア開発に努め，ケンブリッジ大学で68000アセンブリ言語の講義を受けもつ．

1984年，ACORN COMPUTER社に移り，集積回路の設計に従事している．

■監訳者略歴

鈴 木 隆

電気通信大学情報数理工学科卒業．

68000をはじめとする，各種のコンピュータ・システムの開発を手掛け，現在，フリーランスのハードウェア・デザイナーとして各方面で活躍中．

■技術協力

榊 正 憲

参　考　文　献

1）各命令の完全な解説，および68010に関する記述

「M68000マイクロプロセッサ　ユーザーズマニュアル」CQ出版社

2）68020に関する記述

「MC68020 32-Bit Microprocessor User's Manual」PRENTICE-HALL社

3）68000ファミリのハードウェアに関する記述

「MC68000 ファミリ 16/32 ビットデータブック」CQ出版社

4）ACIAに関する記述

「8-BIT MICROPROCESSOR AND PERIPHERAL DATA」MOTOROLA社

68000プログラミング入門

1984年12月5日　初版発行
定価1,700円

著　者　Tim King, Brian Knight
翻　訳　三浦明美
監　訳　鈴木　隆
発行者　塚本慶一郎
発行所　株式会社アスキー
〒107　港区南青山5－11－5住友南青山ビル5F
振　替　東京4-161144
電　話　03—486—7111(代表)

編集担当　佐々木敏久
表紙担当　郷　啓子
印刷　モリモト印刷株式会社

ISBN4-87148-759-8 C3055 ¥1700E

定価 1,700円

ISBN4-87148-759-8 C3055 ¥1700E